RMIT UNIVERSITY

Null Convention Logic Circuits for Asynchronous Computer Architecture

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

Matthew Myungha KIM
Bachelor of Information and Communication Engineering,
Korean Educational Development Institute, South Korea

School of Engineering
College of Science Engineering and Health
RMIT University

October 2019

Declaration of Authorship

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; any editorial work, paid or unpaid, carried out by a third party is acknowledged; and ethics procedures and guidelines have been followed.

I acknowledge the support I have received for my research through the provision of an Australian Government Research Training Program Scholarship.

Matthew Myungha Kim

24 October 2019

Abstract

For most of its history, computer architecture has been able to benefit from a rapid scaling in semiconductor technology, resulting in continuous improvements to CPU design. During that period, synchronous logic has dominated because of its inherent ease of design and abundant tools. However, with the scaling of semiconductor processes into deep sub-micron and then to nano-scale dimensions, computer architecture is hitting a number of roadblocks such as high power and increased process variability.

Asynchronous techniques can potentially offer many advantages compared to conventional synchronous design, including average case vs. worse case performance, robustness in the face of process and operating point variability and the ready availability of high performance, fine grained pipeline architectures. Of the many alternative approaches to asynchronous design, Null Convention Logic (NCL) has the advantage that its quasi delay-insensitive behavior makes it relatively easy to set up complex circuits without the need for exhaustive timing analysis.

This thesis examines the characteristics of an NCL based asynchronous RISC-V CPU and analyses the problems with applying NCL to CPU design. While a number of university and industry groups have previously developed small 8-bit microprocessor architectures using NCL techniques, it is still unclear whether these offer any real advantages over conventional synchronous design. A key objective of this work has been to analyse the impact of larger word widths and more complex architectures on NCL CPU implementations. The research commenced by re-evaluating existing techniques for implementing NCL on programmable devices such as FPGAs. The little work that has been undertaken previously on FPGA implementations of asynchronous logic has been inconclusive and seems to indicate that asynchronous systems cannot be easily implemented in these devices. However, most of this work related to an alternative technique called bundled data, which is not well suited to FPGA implementation because of the difficulty in controlling and matching delays in a "bundle" of signals. On the other hand, this thesis clearly shows that such applications are not only possible with NCL, but

there are some distinct advantages in being able to prototype complex asynchronous systems in a field-programmable technology such as the FPGA.

A large part of the value of NCL derives from its architectural level behavior, inherent pipelining, and optimization opportunities such as the merging of register and combinational logic functions. In this work, a number of NCL multiplier architectures have been analyzed to reveal the performance trade-offs between various non-pipelined, 1D and 2D organizations. Two-dimensional pipelining can easily be applied to regular architectures such as array multipliers in a way that is both high performance and area-efficient. It was found that the performance of 2D pipelining for small networks such as multipliers is around 260% faster than the equivalent non-pipelined design. However, the design uses 265% more transistors so the methodology is mainly of benefit where performance is strongly favored over area. A pipelined 32bit x 32bit signed Baugh-Wooley multiplier with Wallace-Tree Carry Save Adders (CSA), which is representative of a real design used for CPUs and DSPs, was used to further explore this concept as it is faster and has fewer pipeline stages compared to the normal array multiplier using Ripple-Carry adders (RCA). It was found that 1D pipelining with ripple-carry chains is an efficient implementation option but becomes less so for larger multipliers, due to the completion logic for which the delay time depends largely on the number of bits involved in the completion network. The average-case performance of ripple-carry adders was explored using random input vectors and it was observed that it offers little advantage on the smaller multiplier blocks, but this particular timing characteristic of asynchronous design styles becomes increasingly more important as word size grows.

Finally, this research has resulted in the development of the first 32-Bit asynchronous RISC-V CPU core. Called the **Redback RISC**, the architecture is a structure of pipeline rings composed of computational oscillations linked with flow completeness relationships. It has been written using NELL, a commercial description/synthesis tool that outputs standard Verilog. The Redback has been analysed and compared to two approximately equivalent industry standard 32-Bit synchronous RISC-V cores (PicoRV32 and Rocket) that are already fabricated and used in industry. While the NCL implementation is larger than both commercial cores it has similar performance and lower power compared to the PicoRV32. The implementation results were also compared against an existing NCL design tool flow (UNCLE), which showed how much the results of these implementation strategies differ. The Redback RISC has achieved

similar level of throughput and 43% better power and 34% better energy compared to one of the synchronous cores with the same benchmark test and test condition such as input supply voltage. However, it was shown that area is the biggest drawback for NCL CPU design. The core is roughly 2.5× larger than synchronous designs. On the other hand its area is still 2.9× smaller than previous designs using UNCLE tools. The area penalty is largely due to the unavoidable translation into a dual-rail topology when using the standard NCL cell library.

Acknowledgements

It was really long journey. I started with this research group in the middle of 2011 and that is already more than 8 years ago. Sometimes I had to change my study mode to part-time to financially support my family, especially for my two high-school student children. But this research opened up new opportunities for me even before it finished. After four and a half years of study, I was able to start work for a start-up company in Los Angeles working on NCL-based CPU design. Not long after, I joined another Silicon Valley start-up in Mountain View, California to develop a Machine Learning accelerator chip using Null Convention Logic technology and the RISC-V ISA.

Over the eight years, I met numerous great people and they were all excited by and encouraged my research. Especially, I sincerely appreciate my supervisor Associate Professor Paul Beckett with all my heart. Without Paul, I cannot even imagine this research. In every part of my research, Paul waited for me to guide me in the right direction. At every slump, he waited to push me up to the next step. With a lot of patience, he tried to understand all of my circumstances and difficulties.

I would like to acknowledge my co-supervisor Dr. Glenn Matthews for supporting my work plus providing all of the necessary computing environments along with the whole research group at RMIT University – Jing Yu, Andrew Przybylski, Kashfia Haque, Dr. Prashant Dabholkar, Renuka Sovani, Sameh Andrawes, Divya Anshu Bhardwaj, Dr. Tayab Memon and Dr Conrad Jakob, Nguyen Le Huy and Pham Chi Thanh from the RMIT Vietnam campus and Norhazlin Khairudin, Syazilawati Mohamed and my many Malaysian colleagues. I would also like to thank Dr. John Fang and Yinjun Tu, Paul Davis and Dr. Zhe Zhang for all of their help.

The Wave Computing staff - my research consultant Justin Spangaro, Dr. Steve Johnson and Dr. Chris Nicol provided invaluable support for my research in the form of the NELL environment, NCL designs and libraries. Without their help, the research would not have been possible.

I would like to especially thank Karl Fant, the inventor of NCL, for all of his help and guidance along the way, as well as the RISC-V group from UC-Berkeley and Sifive - Dr. Yunsup Lee, Dr. Andrew Waterman, Palmer Dabbelt and Prof. Krste Asanovic and the UNCLE group - Prof. Robert Reese and Prof. Scott Smith.

Dr. Gopal Raghavan, Scott Johnston, Dr. Paul Murtagh, Dr. John Havlicek and my many colleagues from Eta Compute plus Dave Ditzel, Prof. Vojin Oklobdzija, James Burr, Eiji Kasahara, Dr. Tommy Thorn and colleagues from Esperanto Technologies were always supportive of my work.

And finally, a very special thank you to my family, my wife Seonju Kim, daughter Yerim Kim and son James Kim, my father Youngho Kim, my mother Taesu Kim and all of my family in Korea for their unwavering support.

Contents

List of Figures

List of Tables

Dedicated to the three wonderful women in my life, my grandmother, my mother and my wife, for their unwavering belief and constant encouragement

Chapter 1

Introduction

1.1 Overview

For most of its history, computer architecture has been able to benefit from a rapid scaling in semiconductor technology, resulting in continuous improvements to CPU design. More and faster switching devices per unit area have supported increasingly complex hardware operators intended to increase clock rates and improve computational efficiency. During that time, synchronous logic has been the dominant style because of its inherent ease of design and abundant design tools. However, with the scaling of semiconductor processes into deep sub-micron and then to nano-scale dimensions, computer architecture is hitting a number of roadblocks.

Firstly, the power consumption of high-rate clock trees has become a major concern, especially with battery powered, mobile and portable systems, requiring careful management that may include powering down inactive parts of the system. It can be increasingly argued that power per unit performance is becoming even more important to a microprocessor than its overall circuit area. Further, issues such as process variability are making it increasingly difficult to achieve timing closure in complex, high performance systems such as CPU cores.

Asynchronous techniques have been promoted as a potential solution to many of these issues. Asynchronous styles appear to offer many advantages compared to conventional synchronous design, including average case vs. worst case performance, robustness in the face of process, voltage and temperature variability and the ready availability of high performance, fine grained pipeline architectures.

On the other hand, this ability to adapt to operating point changes can lead to the sort of non-deterministic timing behavior that drives some of the many criticisms leveled at

asynchronous design styles in general. A lack of predictable timing can make asynchronous techniques completely unsuited to many applications, particularly those requiring accurate sampling rates (e.g., DSP). However, this issue cannot be considered in isolation. For example, IoT edge nodes and wearable applications are emerging as an important application domain that, as they are mainly operated from battery power, tend to be extremely power sensitive. In these cases, the concept of event-driven data flow can be important when used in conjunction with other techniques such as event-driven sampling [1]. The combined behaviors that, firstly, power will be only consumed when an event occurs and, secondly, that the event is controlled by the onset or presence of data, may open up new application areas in low-power *always on* embedded systems. In contrast, a similar level of clock control in a synchronous design would require careful control of the timing network, probably including sleep management and PLL gating. The requirement to wait until the clock is stable during CPU sleep to wake cycles can cause just as many critical timing issues for event-driven applications. In this case, the lack of a clock or PLL can be an advantage.

This thesis analyses the behavior of a complex asynchronous logic system chosen as a representative case study in the domain of CPU core design targeting event-driven embedded systems applications. Null Convention Logic (NCL) is a type of asynchronous, or clockless digital logic that is attracting attention because its quasi delay-insensitive nature makes it relatively easy to build complex systems without requiring the extensive timing simulation to ensure robust operation under all circumstances. Specifically then, this work explores and analyses the characteristics of an NCL based asynchronous RISC-V CPU and analyses the problems with applying NCL to CPU design.

While it is already the case that modern synchronous designs contain small domains of self-timed logic, these are still bound within the clock domains. As such, the overall system remains fully synchronous. This is not the case for a true asynchronous logic style. This work has focused only on asynchronous logic styles that exhibit two key characteristics: (1) there is no global clock signal of any point in the circuit and (2) logic modules are controlled by handshake signals in which the logic domains create request and acknowledgment or completion signals.

1.2 Motivation and scope

Of the many alternative approaches to asynchronous design, Null Convention Logic (NCL) has the advantage that its quasi delay-insensitive behavior makes it relatively easy to set up complex circuits without the need for exhaustive timing analysis. NCL represents a single coherent logic system with only 27 fundamental operational units (gates) that are sufficient to cover all logic functions. Thus in NCL design it is possible, at least partly, to reuse a conventional synchronous back-end EDA tool flow for its implementation rather than having to rely on the complex full-custom flow commonly used by other Delay-Insensitive design styles.

A number of clockless CPU systems have been designed over the last three decades, with both University and industry groups trying various approaches such as Bundled-Data and Quasi-Delay Insensitive. Although NCL offers a number of advantages as an asynchronous design framework for asynchronous circuits, to date only researchers from the University of Arkansas and an industry group from Theseus Logic have applied it to CPU design. Both of these efforts have targeted similar small 8-Bit (8051-compatible) architectures.

As a result, it is unclear whether the approach is suited to larger architectures such as typical 32-bit CPUs. For example, the performance of NCL depends implicitly on the delay of the completion trees and feedback paths, which grows with word size and therefore will have a significantly greater impact on wider data paths.

For many decades, the embedded CPU domain has been dominated by ARM, MIPS and PowerPC Instruction Set Architectures. However, these are quite complex control-driven architectures and are thus less well suited to asynchronous implementation. RISC-V is a new open Instruction Set Architecture, has a compact Instruction Set and also fewer condition codes and branch delay slots. As a result, a key motivation for this work has been the hypothesis that the RISC-V instruction set may lead to simpler execution control and may potentially be better suited to asynchronous implementation as it matches more closely the data-flow behavior of an asynchronous CPU.

1.3 Research Questions

This research encompasses a study of the characteristics of Null Convention Logic and an analysis of the RISC-V ISA and CPU architecture including its component blocks, in

particular those applicable to a 32-bit CPU core. As mentioned above, although it appears that asynchronous architectures are highly likely to have a place in future computing applications (due to a range of advantages such as near and sub-threshold voltage operation, robustness to variability, low electrical and EMI noise and so on), it is still not clear whether the approach is suitable for high-performance 32-bit CPU architectures. Thus, the research questions that were addressed in this thesis are as follow:

1. What are the challenges for designing NCL based 32-Bit CPU cores? Is NCL technology suitable for high performance or low power embedded CPU design especially for IoT edge nodes applications?

NCL technology is rarely applied to high performance CPU cores, especially for 32-Bit Embedded Processors. As far as we are aware, this is the first time NCL technology has been applied to the design of a 32-Bit CPU core.

2. Can NCL circuits be designed such that they can be implemented on Field Programmable Gate Array devices without requiring extensive and detailed timing constraints to be applied?

Conventional synchronous designs can be easily tested and verified on the commercial FPGA devices and ASIC prototyping on the FPGA is straightforward for those designs because their constraints are almost same and the FPGA devices are intended for that purpose. Asynchronous circuits, particularly Delay Insensitive designs, do not map readily to conventional commercial FPGA devices. This question analyses and contrasts the characteristics of NCL circuits when implemented on commercial FPGA and ASIC.

3. How do asynchronous, data flow processing elements compared to their equivalent synchronous implementations in terms of power, performance and/or area?

Locally connected data flow processing solutions especially arithmetic devices such as adders and multipliers are used widely for many important applications. This question will investigate the comparison between the conventional clocked solutions and NCL for a performance, power and chip area and analyze the advantages and disadvantages of the NCL solution in this domain.

1.4 Research Approach

There is a significant body of prior asynchronous research from both academia and industry, many encompassing microprocessor designs and some focused on other applications. The three major asynchronous approaches that have been studied previously are Bundled-Data, QDI (Quasi Delay Insensitive) and NCL (Null Convention Logic). Note that, although NCL is still strictly a sub-set of QDI, the design approach in this case is quite different from other QDI styles. In this thesis, we differentiate between NCL and non-NCL QDI approaches by referring to the latter as PCHB, as defined by Martin [2]. In general, QDI design is quite complex and typically involves full custom techniques. It also has a range of alternative design and optimization options. In contrast, NCL uses a more *module-based* (or *structural*) approach, employing only a pre-defined library of standard cells. Because generalized QDI design is less structured, it is even harder to reuse the current synchronous EDA tools for its implementation and much prior QDI research has used full custom design rather than automated EDA tools.

In contrast, Bundled-Data is not a QDI style but uses delay elements to guarantee the combinational logic delay is shorter than the handshaking signals such as request and acknowledge. Thus, Bundled-Data is more like conventional clocked synchronous design in nature and normally uses the same standard Boolean gates and therefore is similarly less resistant to technology-related problems such as PVT (Process, Voltage, Temperature) variability.

This research has employed two different implementation strategies: FPGA and ASIC. FPGA implementation supports easy and quick evaluation of the functional behaviour of a system. Various NCL circuits were built and tested on commercial FPGA devices such as Xilinx, Altera and Actel. It was necessary to generate some special FPGA cells for NCL and these were initially tested using a small 8-Bit counter. In the case of an ASIC implementation, it is hard to reuse the foundry supported Clocked-Boolean standard cell library for NCL. It has therefore been necessary to generate a dedicated NCL cell library to test and implement these circuits.

Both Spice and Digital simulation tools have been used in this work. For the cell-level and small block-level designs, Spice simulation was possible using a specially designed cell library. For more complex design such as the CPU, for which Spice simulation would take excessive time, standard Digital simulation tools were used with test-bench models derived from the NCL cell simulation models.

The logic design activities incorporated both UNCLE [3] and NELL design tools [4] into the flow. Using both tools, firstly the CPU components and ultimately the complex CPU core were designed and tested.

Finally, a standard synchronous back-end flow was used for Auto Place and Route followed by Timing and Power analysis. Because those tools are designed for clocked synchronous design, it is not entirely applicable to NCL design, especially for optimization purposes. However, the results were considered to be adequate to compare alternative approaches and to draw general conclusions on the characteristics of the NCL designs.

The RISC-V ISA was chosen for the CPU design. Because RISC-V is open architecture, it has a strong community to support users and there are many open source CPU designs that can be used for comparison purposes. It was also possible to use the RISC-V compiler and test environment for this architecture, as well as their benchmark tests for design verification and to achieve a comparison between various synchronous and asynchronous approaches.

1.4.1 A Note on NCL Signal Labelling

The notation describing the multi-rail system in NCL still very fluid and no single representation has been universally accepted by the research community. This thesis has tended to focus on dual rail organizations and has used the same notation as the work being referred to, where appropriate. This includes variations such as "superscript of 0 and 1" e.g. netname^0, netname^1 and, particularly in the case of UNCLE netlists, t_netname and f_netname. The default notation used for new work is netname/0, netname/1. All of these notations are equivalent and describe the corresponding pair of signal rails in NCL.

1.5 Outcomes and Contributions

1.5.1 Specific Contributions of this work

This research has resulted in the development of a 32-Bit RISC-V CPU core called **Redback RISC**[1] that has been compared to two approximately equivalent industry standard 32-Bit synchronous cores. Further, the implementation results were also compared with the previous NCL design tools (UNCLE), which showed how much the results of these implementation strategies differ. The Redback RISC has achieved similar level of throughput and 43% better power and 34% better energy compared to one of the synchronous cores with the same benchmark test and test condition such as input supply voltage. However, it was shown that area is the biggest drawback for NCL CPU design. The core is roughly 2.5× larger than synchronous designs. On the other hand its area is still 2.9× smaller than previous designs using UNCLE tools.

We also successfully tested the Redback CPU core and many of the component arithmetic blocks on some commercial FPGA devices and have suggested a test methodology in this case. FPGA prototyping is the one of the biggest barrier for asynchronous designs and this methodology has made that easier and clearly showed how commercial FPGA can be used to support NCL circuits. A number of the Dhrystone benchmark tests were demonstrated on FPGA devices using the Redback RISC.

The high performance multi-dimensional design approach especially for the high speed 32 x 32 multiplier also was the first challenge in the NCL design world. High speed multipliers are the basic building block for many application designs such as Finite Impulse Response (FIR) Filters or Fast Fourier Transform (FFT) circuits and also Flow Graph designs like streaming media processing (Video and Audio), Ethernet Packet processing and Cryptography circuits and so on. This research has clearly shown the advantages and trade-offs when using multi-dimensional (e.g., 2-D) NCL designs. Future designers will be able to reuse these results when they consider high performance NCL circuit especially for the Data Flow designs.

A Register File Write-Back Queue design has been proposed in this work and the trade-off between NCL based FIFO and Data-Queue structures shown in terms of performance,

[1]The core was called *Aristotle* when it was presented at the 2nd RISC-V Workshop at UC-Berkeley California in June 2015. The name was later changed to *Redback RISC* to reflect its origin at RMIT University, which uses the name for its competitive teams in both technology and sport. The Redback, or Australian Black Widow, is a highly venomous spider common throughout the whole of the Australian continent.

power and area. Using these comparison results, designers can select the appropriate buffer types for other processor blocks in the NCL based asynchronous microprocessor and system on chip design.

Even though NCL has many technical advantages compared to conventional clocked designs and other asynchronous approaches, much effort is still required to optimize these NCL circuits. The approach described in this work, encompassing both structural level and circuit level optimization of the NCL designs, illustrates a concrete methodology for NCL circuit design with detailed applications such as high speed 32x32 multiplier and 32-Bit high speed microprocessor. This represents a useful design guide for future students and engineers who may wish to apply NCL technology.

1.5.2 Publications

The following publications have arisen directly from this work:

- M.M. Kim, and P. Beckett, "Design Techniques for NCL-based Asynchronous Circuits on Commercial FPGA" in Digital System Design (DSD), Verona (Italy), 2014 17th Euromicro Conference on, pp. 451-458, 2014

- M. M. Kim, K. M. Fant and P. Beckett, "Design of asynchronous RISC CPU register-file Write-Back queue", 2015 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), Daejeon, 2015, pp. 31-36.

- M. M. Kim, J. Kim and P. Beckett, "Area performance tradeoffs in NCL multipliers using two-dimensional pipelining," 2015 International SoC Design Conference (ISOCC), Gyungju, 2015, pp. 125-126.

- J. Kim, M. M. Kim and P. Beckett, "Static leakage control in null convention logic standard cells in 28 nm UTBB-FDSOI CMOS," 2015 International SoC Design Conference (ISOCC), Gyungju, 2015, pp. 99-100.

- Matthew M. Kim, Karl M. Fant, Paul Beckett, "Aristotle. A Logically Determined (Clockless) RISC-V RV32I [5]", 2nd RISC-V Workshop, June 29-30, 2015, (non-refereed), The International House, Berkeley, CA

1.6 Thesis Organization

The remainder of the thesis is organized as follows:

Chapter 2, Background and Literature Review:
This chapter provides firstly an overview of the background of asynchronous digital logic technology and the detail of the major asynchronous design styles (Bundled-Data, Quasi-Delay Insensitive and Null Convention Logic). It introduces the theory of NCL and then discusses it in more detail as this is the technology used for this research. Previous research related to asynchronous logic design is described along with their implementation on commercial FPGA devices. It presents the design history and trends for asynchronous CPU architectures. Finally, the RISC-V Instruction Set Architecture is introduced that was used for the CPU core design in this research.

Chapter 3: NCL Design Methodology:
This Design Methodology chapter describes the design flow used to create the NCL circuits and also introduces the NCL design tools "UNCLE" and "NELL". NCL based ASIC design methodology is discussed from the NCL gate design to the semiconductor Back-end chip implementation. Additionally, the methodology of NCL circuit implementation on commercial FPGA devices introduced including their FPGA cell library designs. The area (in terms of gate count) of the networks generated by the alternative tool flows are compared using a number of small benchmark circuits.

Chapter 4, NCL Circuit Design for CPU:
This chapter introduces the standard CPU components from which the NCL based 32-Bit CPU core has been formed, as well as their circuit-level optimization techniques. Firstly, a number of NCL Adder designs are introduced and compared against their specifications. Then a 32-Bit NCL Multiplier circuit is built and analysed along with a 32-Bit NCL Barrel Shifter that has been used for this NCL based RISC-V CPU design. Finally, two additional important CPU blocks are discussed, the NCL Register File and NCL Program Counter.

Chapter 5, Redback RISC Design and Optimization:
The details of Redback RISC core are explained in this chapter plus its comparison results between this and other synchronous RISC-V cores. Firstly, the RISC-V ISA details are described followed by the specific implementation in this work. This chapter also includes details of

the design methodology that was used for the CPU core comparison and also the test and verification methodologies. Finally, it shows the CPU comparison and benchmark test results.

Chapter 6, Conclusion and Future Work:
This final chapter concludes this thesis and presents the future work.

Chapter 2

Background and Literature Review

This chapter presents an overview of asynchronous design technology and reviews the current literature in the domain. The advantages of asynchronous technology are discussed along with current trends in asynchronous design. Three major asynchronous technologies are included: Bundled-Data (BD), Quasi-Delay Insensitive (QDI) and Null Convention Logic (NCL). Their design styles are shown and their specifications compared.

Null Convention Logic is then discussed in more detail as it is the technology used for this research. Prior asynchronous CPU design activities are reviewed. And finally, we briefly introduce the RISC-V Instruction Set Architecture that was used for the CPU core design in this research.

2.1 Asynchronous Technology

Conventional synchronous design uses a centralized clock signal to control the data flow through combinational logic between edge-triggered register stages. In contrast, asynchronous design relies on localized handshaking signals (e.g., Request and Acknowledge) to control its data flow. Of the three types of asynchronous models in current use (Bounded Delay model, QDI and NCL), NCL has the least sensitive delay constraint. In the Bounded Delay model (such as micro-pipelines), delays in both gates and wires are bounded and delays are evaluated based on worst-case scenarios to avoid signal hazard conditions. QDI is constrained by the isochronic fork, which assumes uniform wire delays and uniform switching thresholds at the inputs to the gates associated with the forking branches. The latter is often difficult to achieve in the face of PVT variability. NCL is constrained by orphan paths, nets that do not contribute to the output and are therefore not part of the completion network, and which are therefore considerably easier to manage and/or remove than the isochronic fork.

Null Convention Logic is based on threshold logic gates exhibiting internal state holding in each gate except, of course, for the TH1x set of gates. It is a symbolically complete logic system that exhibits delay insensitive behaviour within some minor constraints. Unlike other DI techniques, NCL gates rely on one simple timing assumption: that the feedback path for its state holding elements must be faster than the forward delay through the gate. This limitation is usually straightforward to overcome.

Input *completeness* requires that the output signal of the gate may not transition to DATA until all inputs have transitioned to DATA. Similarly, the output will not transition to NULL until all inputs have reached NULL. Input completeness is achieved by adding the value NULL to the basic logic values (TRUE and FALSE), to represent a status of "no data". Output Data will only be valid when all input signals have transitioned from NULL to DATA. An NCL circuit consists of an interconnection of primitive modules known as M-of-N threshold gates with hysteresis. All functional blocks, including both combinational logic and storage elements, are constructed out of these same primitives. To represent these three values in this work we use a dual rail coding style which expresses the value '00' (NULL), '01' FALSE and '10' TRUE. A NULL wave front separates two DATA wave fronts.

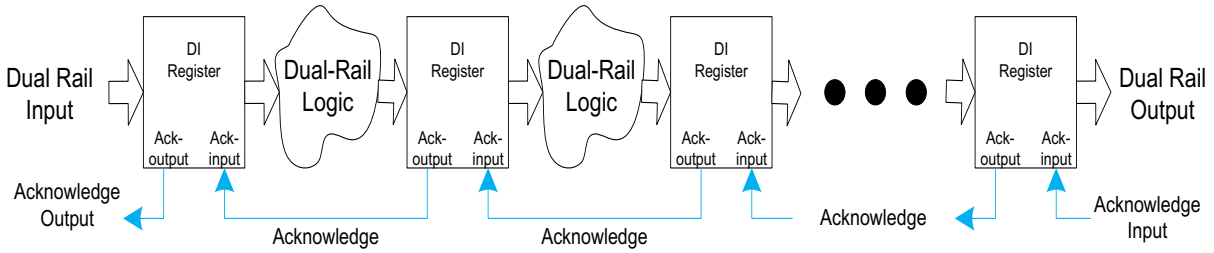

Figure 1: Delay Insensitive Model

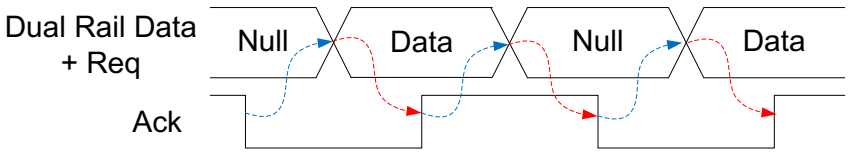

Figure 2: Delay Insensitive Protocol

Figure 1 shows the data flow of the delay insensitive model. The Request signals are embedded within the dual-rail data input and the circuit has separate Acknowledge signals. Figure 2 is an illustration of the four-phase delay insensitive handshaking protocol and shows the handshaking timing flow between dual-rail input data and single-rail acknowledge signals.

This protocol is very robust, the sender and receiver of the data can communicate reliably regardless of delays in the wires.

2.1.1 Why Asynchronous technology?

As mentioned in the previous section, synchronous logic has been the dominant style for most of the history of computer architecture because of its inherent ease of design and abundant design tools. However, the scaling of semiconductor processes is causing a number of serious obstacles, including escalating power and increasing variability due to process, voltage and temperature (PVT). Robustness against variability is one of the biggest benefits of using asynchronous technology, particularly Quasi Delay Insensitive styles such as NCL. The requirement to maintain sufficient timing margins across all operating conditions means that synchronous logic can be very sensitive to PVT variations. Thus, as technology moves further into the deep sub-micron region it becomes harder to scale down the supply voltage on synchronous systems. In contrast, Delay Insensitive circuits will almost always work regardless of the variation in delay of its component parts. The performance will be slow when the supply voltage is near-threshold or sub-threshold but will still be logically correct. It is worth noting that if extreme voltage scaling (VS) is applied, wire/gate delay ratios change in turn affecting the internal forks of NCL gates perhaps requiring careful gate design. Straightforward dynamic power management techniques become available in DI systems. We can increase the supply when higher performance is required or reduce it to save power without having to consider timing. As power consumption is proportional to the square of the supply voltage, these types of techniques will be important to power sensitive application such as battery operated mobile devices or IoT.

There are also some basic design style considerations that separate synchronous and asynchronous design. For example, in synchronous RTL design, it is common to pass signals through multiple paths and to *select* (multiplex) the "correct" signal at the end of the path. Thus, all paths continually contribute to the overall switching activity in the system. In contrast, asynchronous design and especially NCL, the circuit style is naturally *deselect* (demultiplexer) oriented, which means the sub-blocks of a logic system will operate only when their action is required otherwise no switching activity occurs. In the NCL domain, this is called signal *steering*. It is possible to design similar networks in a synchronous system but it requires additional resources and careful timing analysis. In this way, asynchronous techniques can

present more power saving opportunities when the circuit has many execution parts that work selectively–under data control, for example. This is certainly the case in CPU design where the active paths within the core are directly controlled by the current instruction.

As introduced in Chapter 1, asynchronous designs exhibit event-driven behaviour. Power will be only consumed when the event is happening. To achieve the same effect in a synchronous system would require complex PLL control and it would be necessary to wait until the clock is stable each time the CPU moves from its wake and sleep states. The delay time for the PLL would therefore be critical for event-driven applications. But in asynchronous case, as there is no clock and no PLL sub-system, event-control will be much more efficient than in synchronous design.

There are other potential advantages to asynchronous design styles introduced in [6] and [7].

- Low power consumption: due to fine-grained clock gating, no global clock distribution and their clock drivers and zero standby power consumption;
- High operating speed: absence of clock skew, average-case performance, operating speed is determined by actual local latency values rather than global worst-case latency;
- Less emission of electro-magnetic noise: as the local handshake signals tend to "tick" at random points in time, asynchronous circuits exhibit a more distributed noise spectrum and low peak noise;
- Better composability and modularity: because of the simple handshake interface and the local timing, additional modules can be added regardless their internal delay time.

2.1.2 Asynchronous Design Trends

Asynchronous technology has been studied over many years in both academia and industry. Asynchronous technology started very early in the history of computing, even predating clocked systems. A number of the early generation computers were implemented completely asynchronously, for example the *ILLIAC II* in 1962 [8].

Currently there are three dominant asynchronous technologies: Bundled-Data, QDI and NCL. The details of each technology will be introduced in Section 2.2.

2.1.3 Recent Asynchronous Activities and Applications

As mentioned, asynchronous techniques have been around for many years. However, in recent times there seems to have been a revival in interest and a number of industry groups have been formed to commercialize asynchronous technology, some more successfully than others. Table 1 highlights some of these companies and their technology and lists their main application. A few of the more important developments are described below.

Achronix [9] is focused on the commercialization of asynchronous FPGA technology. Their picoPIPE methodology adds a register block before and after the combinational FPGA reconfigurable blocks which reduces the propagation delay of the combinational blocks and increases the overall performance (i.e. results in reduced cycle times). Their performance is generally around three times faster than conventional Clocked Boolean Logic based FPGA [9]. They have basically used PCHB QDI technology for their register transfer implementation. Further details of this technology will be introduced in Section 2.3.2.

Fulcrum Microsystems was set up to exploit high-speed asynchronous packet switch solutions especially for Ethernet Packet processing devices. They also developed a RAM based crossbar switch with a packet scheduler. The company was acquired by Intel in 2011, which then announced a range of high performance network server solutions containing Fulcrum's asynchronous switch technology. Intel has now developed the "*Loihi Asynchronous Neuromorphic Research Chip*" [10] for their so-called Neuromorphic applications. This was designed using Bundled-Data techniques.

Tiempo [11] is a French asynchronous design company that has developed an asynchronous synthesis tools called the Asynchronous Circuit Compiler (ACC). Tiempo currently develops security related products using its asynchronous technology.

Eta Compute [12] started in 2015 and their application is machine intelligence in mobile and edge devices and IoT. The company is using a technology called DIAL (Delay Insensitive Asynchronous Logic) and their focus is on low power ASIC design.

Finally, Wave Computing have developed an asynchronous design tool called NELL (NCL Equation Logic Language). The NELL suite includes a programming language, compiler and simulator aimed specifically at Null Convention Logic technology. NELL was used for the asynchronous CPU design described later in this thesis.

Table 1: Asynchronous Company List

Company	Technology	Comments
Achronix	QDI, asynchronous FPGA	QDI based FPGA. Main applications are high performance graphics and network packet processing. DI technology robust against PVT variations
Intel (Fulcrum Microsystems)	QDI, Bundled data, asynch. packet switch	Focuses on asynchronous packet switching with switch-matrix solution for high-speed SoCs. Acquired by Intel in 2013. Loihi Asynchronous Neuromorphic Research Chip uses BD
Handshake Solution	Bundled data processor	Spin-off from Philips. Designed a low-power event-driven 8051 compatible asynchronous CPU then an Asynchronous ARM processor. Now re-merged with Philips/NXP)
Theseus Logic	NCL, processor, smart sensors	First company to commercialize NCL technology. The inventor of NCL, Karl Fant was the co-founder of the company. Later sold to Camgian Systems
Tiempo Secure	QDI, processor, security solutions	French company, with 16-Bit microprocessor. Main market is for security chip solutions —passports and credit cards. Event-driven chip design methodologies
Wave Semiconductor	NCL, asynchronous multi-processor	A start-up company in Silicon Valley, 2009, with Karl Fant as one of the founding members. Used NCL technology to design a multi-core asynchronous processor aimed at the AI market. Developed NELL design tool to support that work. Now called Wave Computing
Eta Compute	DIAL, Machine learning, edge devices	Started in 2015. Machine intelligence applications in mobile and edge devices and IoT. DIAL (delay insensitive asynchronous logic) Focus on Low Power ASIC design
Octasic	synchronous / asynch. Low power DSP	A DSP design company using asynchronous technology. Their DSP solution has very low power compared to the commercial clocked Boolean logic solutions

2.1.4 Asynchronous Tools

In case of Clocked Boolean Logic (CBL) design, we can easily find an appropriate tool set for all types of applications such as ASIC, FPGA, DSP and CPU designs. In contrast, it is hard to find suitable tools for asynchronous technology. This section introduces several existing asynchronous design tools.

- TiDE/Haste tool: TiDE stands for Timeless Design Environment and Haste is a high level design entry tool from Handshake Solution [13]. The tools execute behavioral synthesis and generating Verilog net-list. These tools are based on Bundled Data technology

- Balsa system: Balsa was designed by researchers at the University of Manchester [14] and is a CSP (Communicating Sequential Processes)-like language based on the *Tangram* VLSI language. The Tangram programming language was developed by Philips Electronics and Eindhoven University of Technology. It can be transparently compiled into an intermediate representation (between the language and gate level implementations) called handshake circuits.

- UNCLE: UNCLE [15] stands for the Unified NULL Convention Logic Environment and was designed at the University of Mississippi. UNCLE employs commercial synthesis tools to produce a gate-level net-list with primitive logic gates that UNCLE has internally and UNCLE transformed the net-list into NCL net-list by their mapping flow. The details of UNCLE will be discussed in Section 3.1.1.

- Proteus: Proteus is an asynchronous ASIC CAD flow, developed by TimeLess Design Automation based on work at the University of Southern California that was bought by Fulcrum Microsystems in 2010 [16]. Proteus is based on a proprietary cell library of domino logic and asynchronous control cells. It comprises a small collection of gates that have been sized, laid out and characterized as individual library cells designed to implement robust high-performance pipelined circuits. The Proteus flow leverages both synchronous synthesis and place-and-route tools and, as a starting point, supports legacy RTL designs as well as Fulcrum's proprietary high-level language based on Communicating Sequential Processes (CSP).

- NELL: the NCL Equation Logic Language (NELL) was developed by Wave Computing and is an independent NCL description language (and also compiler, simulator and debugger). NELL is designed as an effective means of expressing NCL circuits. The details of NELL will be discussed in Section 3.1.2

- ACC (Asynchronous Circuit Compiler): ACC is the asynchronous synthesis tool from Tiempo which automatically generates asynchronous and delay-insensitive circuits from a model written in a standard hardware description language. ACC takes as its input a description written in System Verilog, which is ideally suited for high-level modeling of clockless circuits, and generates as output a gate-level net-list in standard Verilog format [11].

2.2 Asynchronous Styles: Bundled Data, QDI and NCL

In this section, the three major asynchronous design technologies are discussed and the details of the NCL design technologies are introduced including its brief history and the theory of NCL gate designs.

2.2.1 Bundled Data

The term bundled-data refers to a situation where the data signals use normal Boolean levels to encode information, while separate request and acknowledge wires are bundled with the data signals [6]. Bundled-Data uses two alternative handshaking protocols, 4-phase and 2-phase. Four-phase bundled-data most closely resembles synchronous design and so, compared to other asynchronous design styles, is the most straightforward to implement and is also the most amenable to the use of existing (synchronous) development tools. The 2-phase bundled data technique was introduced with Sutherland's Micro-pipelines [17]. 2-phase bundled-data protocol should lead to faster circuits than 4-phase but the circuit will be more complex to implement [6]. As introduced previously, many previous asynchronous designs used Bundled-Data technology such as the University of Manchester group, Handshake Solution, etc.

In Bundled-Data design, we can reuse most of the standard Boolean gates and also synchronous EDA design tools except for a few logic elements such as Muller C-Elements. Its main drawback is that it is less robust and it still needs critical timing analysis to meet the timing requirements arising from PVT variability.

2.2.2 Quasi Delay Insensitive

while Delay insensitive (DI) is a robust circuit design solution, the class of true delay-insensitive circuits is unfortunately rather small. Only circuits composed of C-elements and inverters can be completely delay-insensitive [6]. Circuits that are delay-insensitive with the exception of some carefully identified wire forks are called quasi-delay-insensitive (QDI). Such wire forks, where signal transitions occur at the same time at all end-points, are called *isochronic*. Typically, these isochronic forks are found in gate-level implementation of basic building blocks where the designer can control the wire delays. At higher levels of abstraction, the composition of building blocks would typically be delay-insensitive. In delay-insensitive circuits both the data and its validity are encoded on the same N wires rather than as separate request and data wires therefore no timing assumptions are needed. This increases its robustness to PVT variations, and reduces the amount of timing verification required [7]. In QDI, the asynchronous blocks can be ultimately decomposed into a hierarchical network of leaf cells, where a leaf cell is the smallest block that communicates with its neighbors via a channel. This significantly reduces the manual efforts required and facilitates the use of conventional synchronous back-end tool flows. The technique was pioneered by Alain Martin at Caltech and has been applied to several asynchronous microprocessors [7]. In recent times, the QDI asynchronous design style has been widely used. Specific examples have used Domino logic style to create the PCHB (Pre-Charge Half Buffer). However Domino logic is limited to full-custom design flow because of its reduced noise margin and because its timing assumptions are not currently supported by semi-custom design tools [7]. Some recent asynchronous design flows embedded Domino logic within a pipeline template and tried to use conventional ASIC flow for their back-end implementation using template based leaf cells. Domino logic potentially offers higher performance compared to normal static circuit design but as just mentioned, has lower noise margin and is less robust compared to its static counterpart, especially when the supply voltage is near or at sub-threshold.

2.2.3 Null Convention Logic

In this section, we will introduce the behavior of Null Convention Logic (NCL) technology and its historical background[1]. Null Convention Logic (NCL) is a complete and coherent logic of majority logic operators (termed “threshold” gates in NCL) with state holding

[1]The basic theory of NCL circuit design is explained in the two NCL books [18] [19] and also in the many NCL patents by Fant and others

behavior. Its operators are sufficient to implement a complete system with minimal non critical delay considerations purely in terms of logical relationships (no clock). Because NCL systems are effectively logically determined they are simpler to design and more robust than other asynchronous design methods which involve complex timing analysis such as isochronic forks or matched delay lines. Null Convention Logic is still a member of the QDI class, but the generalized QDI techniques mentioned earlier uses a heterogeneous set of components: registers, handshaking of various flavors, specialized dual rail circuits and some very complex design methods. In contrast, NCL creates a single coherent logic system with only 27 fundamental operational units (gates) which are sufficient to cover all logic functions. Therefore NCL is more amenable to the reuse of a conventional clocked back-end EDA tool flow for its implementation.

2.2.3.1 A Brief History of NCL

The first NCL patent [20] was filed by Fant and Brandt in 1994 and in 1996 full details were published in [21]. Later, in 2005, all of the basic technology background and along with detailed application designs were presented in [18]

Karl Fant had invented the technology while at Honeywell Systems in Minnesota, USA. In 1990, he left Honeywell and started a company called Theseus Research [22]. In 1996 Theseus Logic was spun out of Theseus Research to commercialize NULL Convention Logic (NCL). That company was acquired by Camgian Networks Inc in 2007. In 2010, Karl co-founded Wave Semiconductor, a fabless start-up in California intended to commercialize a high performance clockless multi-processor design based on NCL. The company later changed its name to Wave Computing and started to focus on AI applications. In 2014, the foundation patents for Null Convention Logic began to expire and are now in the public domain, so that the basic technology can now be used with few limitations.

The most recent enhancement to the basic NCL technology has been by Scott Smith and Jia Di from the University of Arkansas. They developed an ultra-low-power variation that incorporates multi-threshold CMOS transistors called Multi-threshold Null Convention Logic (MTNCL), or alternatively Sleep Convention Logic (SCL) [19].

2.2.3.2 What is NCL?

NCL is an acronym standing for Null Convention Logic. The term *Null* implies "empty" or "No Data", which expresses the added empty sequences between Data sequences. The same concept is generally termed "empty" in other asynchronous methodologies.

Figure 3 shows the expression diagram for NCL [18]. A data path presents successive data values by monotonically transitioning between "Completely Data" and "Completely Null".

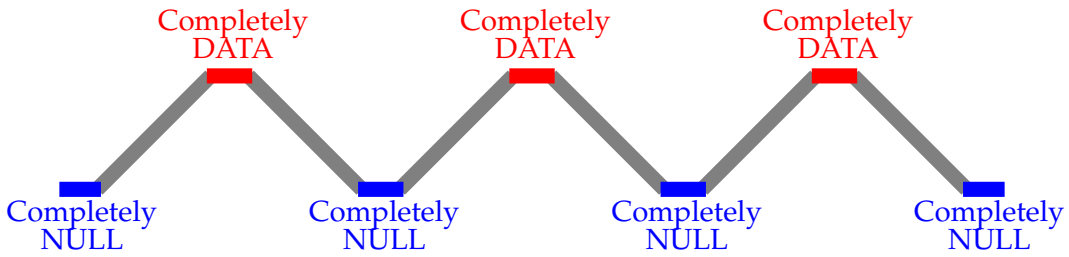

Figure 3: Expression Diagram of Null Convention Logic - redrawn from [18]

- **Ring Oscillator**

A conventional synchronous Boolean circuit generally uses a crystal oscillator as a clock source to ensure that it is stable and accurate. Because clocked Boolean logic always uses worst-case timing assumptions, it requires a very accurate clock source. In contrast, Delay Insensitive circuits do not need these sorts of accurate clocks as the handshake control will automatically adjust to operate at the minimum delay allowed by the technology and its environment.

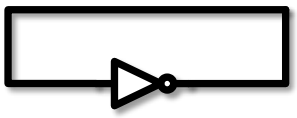

Figure 4: Inverter Oscillator

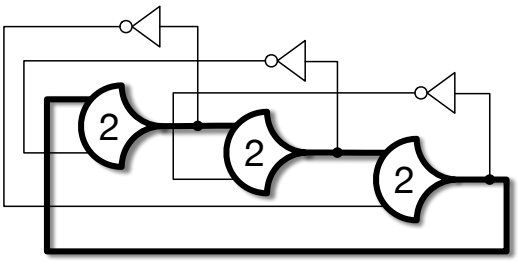

Figure 5: NCL Ring Oscillator

Figure 4 shows the Inverter-Oscillator of a generalized Boolean logic circuit that oscillates automatically after power up. Figure 5 illustrates the NCL Ring Oscillator with TH22 gate and inverter, which is equivalent to the C-element in other asynchronous design styles. This

oscillation imparts *liveness* to the whole NCL circuit. The concept of liveness is borrowed from other asynchronous domains such as Petri nets and asynchronous (event-driven) programming. Informally, it implies that a complete set of inputs will, eventually, result in a complete set of outputs. Here, the term is used to describe the equivalent to a clock in synchronous systems i.e., the mechanism that causes and sustains continuous operation. For example, the controlled increment function of a Program Counter will cause a CPU to continuously perform its fetch—-decode—-execute cycle.

• **Completeness and Hysteresis**

From Figure 3, NCL inserts a NULL state between two DATA states. The output of the NCL circuit transitions to DATA only when its input is "Completely Data". Conversely, an output will transition to NULL only when its corresponding inputs are "Completely Null". It maintains its output when its inputs are in neither of these conditions.

Figure 6 shows an example hysteresis diagram of a TH35 NCL gate. When the number of input data values matches the threshold (in this case 3), the threshold operator transitions its output to DATA. Similarly, when all input values become NULL the threshold operator transitions its output to NULL. Thus, unlike other asynchronous technologies, NCL imposes a C-element-like behavior on *all* its gates and, in addition, all gates exhibit hysteresis. Using these monotonic transition functions, glitch–free circuits can be implemented.

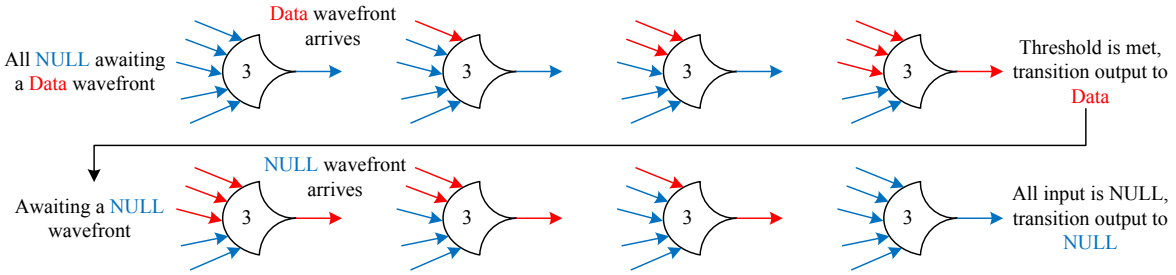

Figure 6: Hysteresis Diagram of TH35 - redrawn from [18]

Figure 7 shows how the NCL operators are formed [18]. The operators enclosed in red are the same as Boolean OR gates such as OR2, OR3, OR4 and OR5 while the operators in the green region are equivalent to the C-element. Finally, the operators within the blue shaded area are the special NCL operators and the number inside the operator is the threshold for the

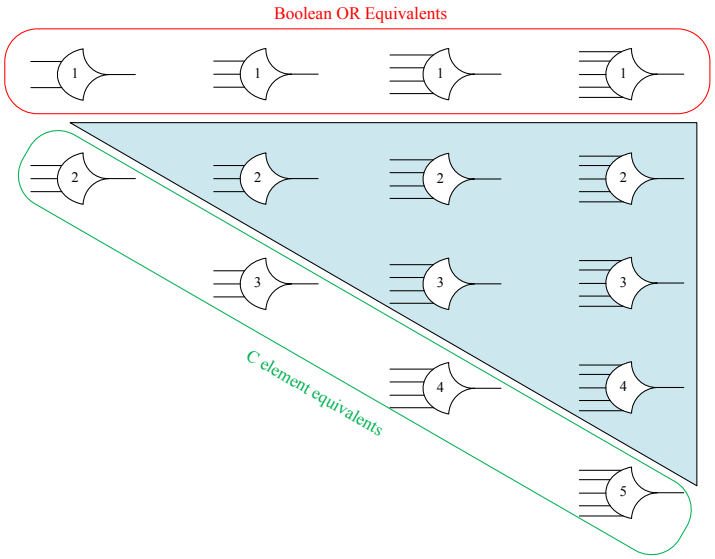

Figure 7: NCL Operators - redrawn from [18]

number of indicated inputs [2]. There is no equivalent of the logical inverter or the Boolean AND operation in the set of NCL operators.

- **Demux Oriented Fan-out/Fan-in**

As was discussed in Section 2.1.1, NCL circuit design tends to be naturally decoder/demultiplexer (demux) oriented. In this topology (Figure 8), only the selected function "fires" and generates a result that proceeds through to the output. In the corresponding synchronous (Mux oriented) organization (Figure 9), all of the functions are driven by their input signals and the desired result is then selected. This clearly represents a waste of power as it generates unnecessary switching activity [18].

2.2.3.3 NCL Gate Design

An NCL circuit consists of an interconnection of primitive modules known as M-of-N threshold gates with hysteresis. All functional blocks, including both combinational logic and storage elements, are constructed from these same primitives. To represent these three values in this work we have used a dual rail coding style which expresses the value '00' (NULL), '01' FALSE and '10' TRUE, with a NULL wave front separating two DATA wave fronts. Unlike other DI styles, NCL uses only one type of state holding gate and NCL uses this threshold gates as its basic logic element [23].

[2] There are NCL gates outside this scheme such as THXOR, THAND etc

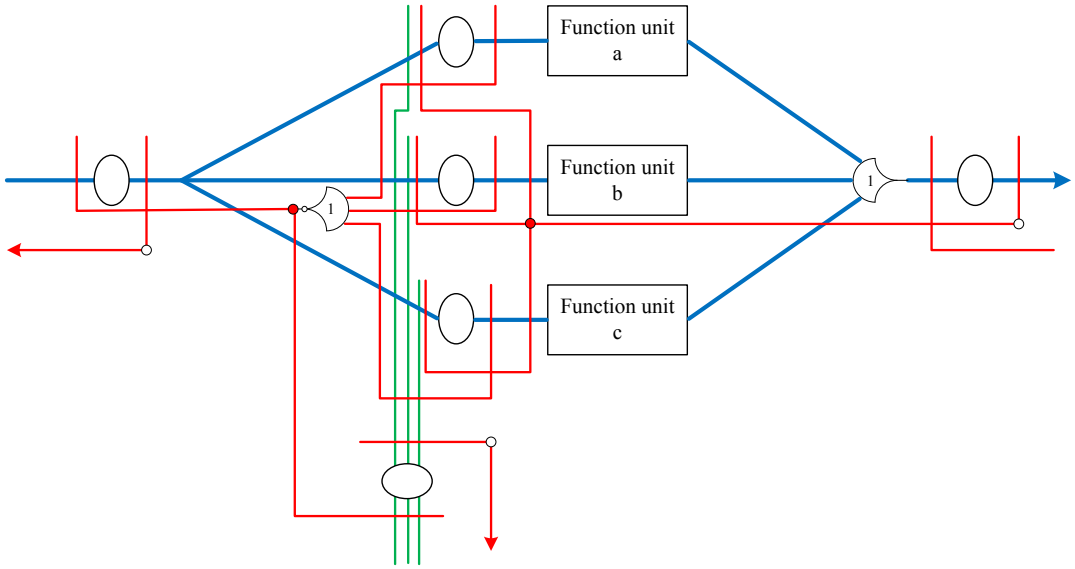

Figure 8: Demux oriented - redrawn from [18]

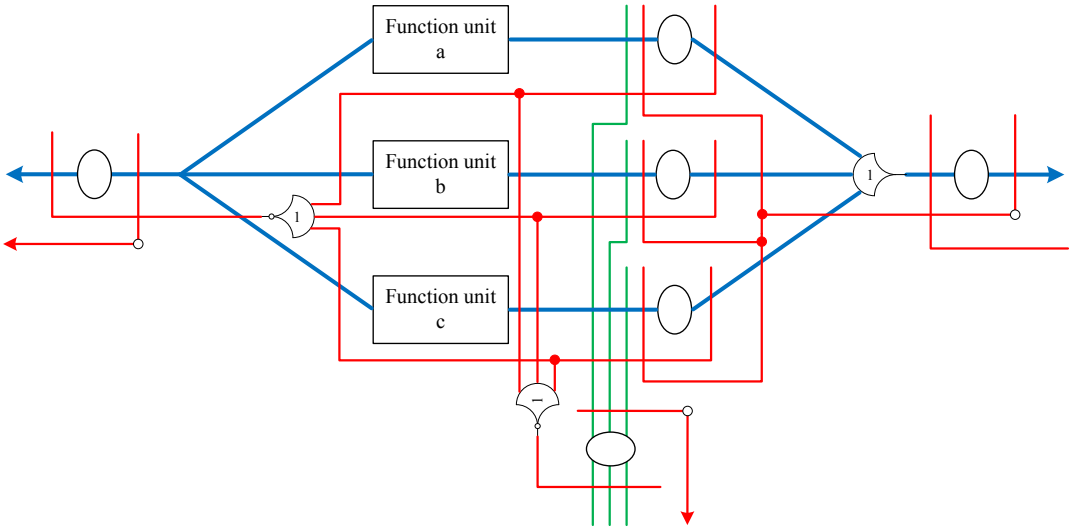

Figure 9: Mux oriented - redrawn from [18]

As NCL threshold gates are state-holding and designed to exhibit hysteresis they comprise circuit blocks for Set, Reset, Hold0 and Hold1 (Figure 10(a)). The Set and Reset functions have their usual functions while the Hold functions (Hold0 and Hold1) implement hysteresis [23], [24]. Once the set function becomes true the output is asserted and the output then remains asserted via the Hold1 network until all inputs return to NULL [24], [25].

Figure 10(b) illustrates a basic THmn gate, where $1 \leq m \leq n$. This gate model has n input signals and a single output. At least m of n inputs must be asserted before the output is asserted, as indicated by 'm' on the symbol in Figure 10(b). An example of a second class

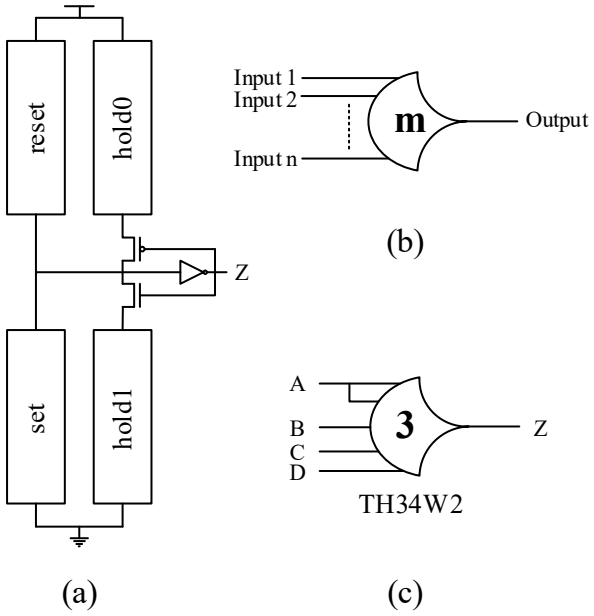

Figure 10: Structure of NCL Gates

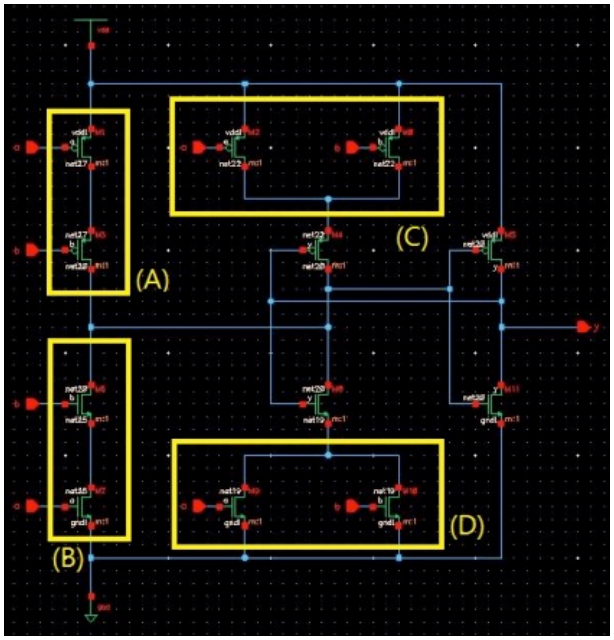

Figure 11: TH22 Circuit Design

of (weighted) threshold gate is shown in Figure 10(c) [23]. Here, the function is expressed as THmnw1w2..wR, where the weight wR is in the range $M \geq wR \geq 1$ [3].

In the transistor level schematic of a TH22 gate (Figure 11), Block (A) represents the reset part, Block (B) the set part, while Block (C) and (D) are the Hold0 and Hold1 respectively. In this example, the set equation is $(a.b)$ and the $Hold1$ equation is given by $(a + b)$. Reset is the complement of $Hold1$: $(\overline{a}.\overline{b})$ and $Hold0$ is complement of set: $(\overline{a} + \overline{b})$. In general terms, the output Boolean equations can be described as: $Z = set + (\overline{Z}.Hold1)$ where Z is the current output and $\overline{Z}$ is the previous output of the gate. The complement of Z is Z' and also can be described as: $Z' = reset + (\overline{Z'}.Hold0)$ [24].

Table 2 shows the 27 Fundamental NCL gates and their equivalent Boolean Equations. The equations do not express the NCL Hysteresis functions but only the Set functions of Figure 10(a). It can be seen that the table 7 actually contains 37 cells. Although Fant [18] has proven that 27 gates are sufficient to synthesise all Boolean logic equations, the ten additional cells cover commonly used variations such as register cells with initialisation (set/reset) inputs and individual buffer/inverter cells.

[3] w1 is omitted in the gate definition as it is redundant.

Table 2: Fundamental NCL Gates

NCL Operators	Boolean Equations
TH12	A + B
TH22	AB
TH13	A + B + C
TH23	AB + BC + AC
TH33	ABC
TH23w2	A + BC
TH33w2	AB + AC
TH14	A + B + C + D
TH24	AB + AC + AD + BC + BD + CD
TH34	ABC + ABD + ACD + BCD
TH44	ABCD
TH24w2	A + BC + BD + CD
TH34w2	AB + AC + AD + BCD
TH44w22	ABC + ABD + ACD
TH34w3	A + BCD
TH44w3	AB + AC + AD
TH23w22	A + B + CD
TH34w22	AB + AC + AD + BC + BD
TH44w22	AB + ACD + BCD
TH54w22	ABC + ABD
TH34w32	A + BC + BD
TH54w32	AB + ACD
TH44w322	AB + AC + AD + BC
TH54w322	AB + AC + BCD
THXOR	AB + CD
THAND	AB + BC + AD
THCOMP	AC + BC + AD + BD

2.3 Asynchronous Design on a Commercial FPGA

One of the difficulties of all asynchronous logic approaches is their unsuitability for use in standard FPGA devices using a conventional tool chain. While there have been a number of attempts to implement asynchronous design on commercial FPGA Look-up Tables (LUT) [26]–[33] these have tended to require either special steps to be inserted into the tool flow, or for the circuit's timing to be (manually) constrained such that it complies with the applicable delay model. For example, a LUT-based method has been described in [26] but its Delay Insensitive approach mandates strict constraints to meet the balanced isochronic fork requirement. Further, the LUT based design is limited to a specific FPGA vendor. In a similar manner, the asynchronous logic implemented in [27] (targeting a Xilinx LUT) and [31] requires complicated and strict timing constraints to ensure that the synthesized logic meets its target delay

time. The asynchronous LUT-based technique proposed for sensor network design by Liu et al. [28] also requires very careful timing constraints and delay calculations. Although the XBM controller implemented in [29] did achieve hazard free asynchronous behaviour, the approach would be difficult to apply in a general case and is inefficient. Ho et al. [30] also implemented QDI logic but this had to be designed by hand to carefully control the delay time. Brunvand [33] also used an FPGA to implement a library comprising bundled data modules. As these were created using specific Actel macro blocks, the approach will not be generally applicable.

In response to these problems, there have been a number of proposals for building specialized LUT structures that directly support NCL gate implementations. The structure suggested in [23] is very similar to the commercial Xilinx Virtex-6 6 input LUT, while the special reconfigurable cell for NCL described [34] is only applicable to their special Atmel FPGA organisation rather than to general commercial FPGA devices.

2.3.1 Dedicated NCL FPGA Devices

A Reconfigurable NCL Logic Element (LE) was proposed in [23], [35] and Figure 12 shows the architecture of that LE. This architecture maps the combinational logic onto the Look-Up Table (LUT) and develops the hysteresis function (Hold Logic) using a weak inverter and includes initialization configuration registers.

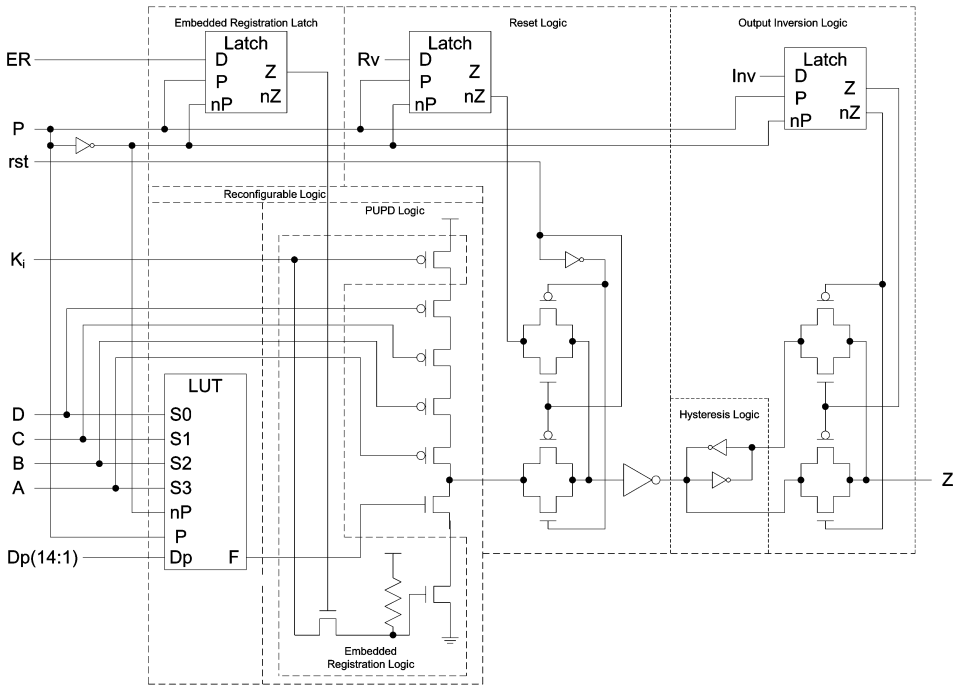

Figure 12: Reconfigurable NCL LE with extra Embedded Registration [23], [35]

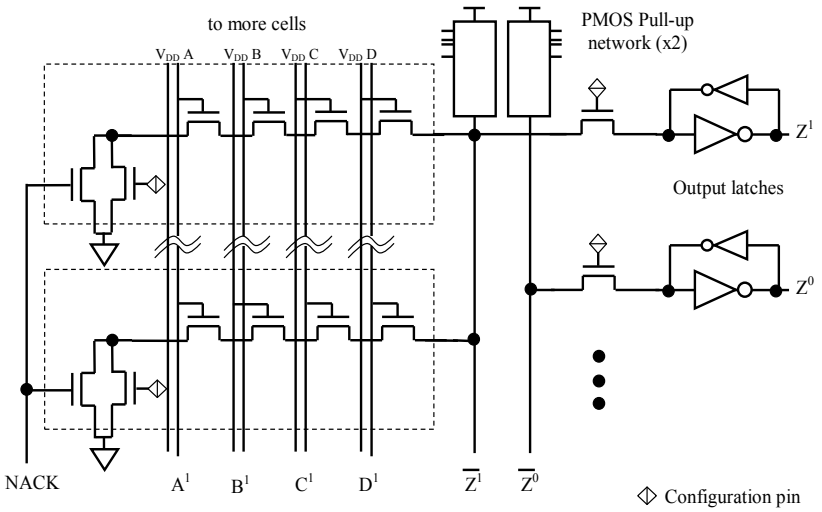

Figure 13: Dual-Rail Reconfigurable Logic Block [36]

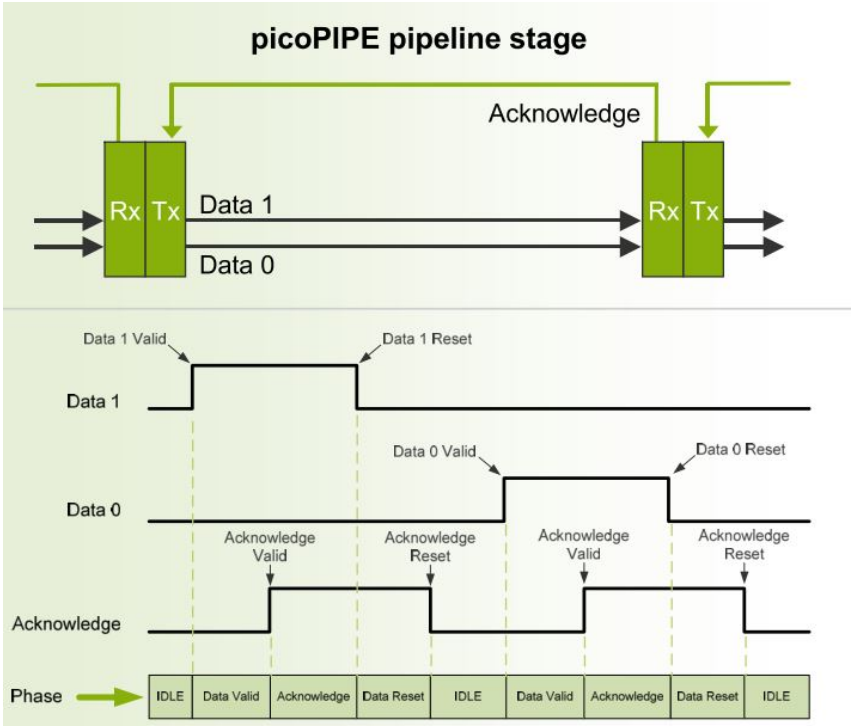

Figure 14: pipoPIPE Technology - Achronix [37]

A simplified block diagram of the dual-rail LUT described in [36] is shown in Figure 13. Because this is a dual-rail block it has two output ports (labeled Z^0 and Z^1 in the figure). This circuit also includes an acknowledge output port for asynchronous handshaking.

2.3.2 Achronix

Achronix [9] is currently one of the more successful firms offering asynchronous products. Using their very fine-grained pipelining approach, the Pico-Pipe [37], it is possible to

outperform conventional reconfigurable systems. Figure 14 shows the basic topology of the Achronix picoPIPE. A register block has been added before and after the combinational FPGA reconfigurable blocks which reduces the propagation delay of the combinational block and increases the performance, as measured by the overall cycle-time. Their published performance results are generally three times faster [9] than for a conventional Clocked Boolean Logic based FPGA.

2.4 Asynchronous CPU Design History and Trends

Asynchronous CPU history almost tracks asynchronous history because most of the asynchronous research groups have tried to use a simple CPU as the example application to demonstrate their technology. Even today, CPU design is still a major application domain for asynchronous logic and, with the rise in importance of embedded systems, is likely to remain so. This analysis splits asynchronous CPU architectures as 3-Generations, which are 1) Early Generation Asynchronous CPU Machines, 2) Asynchronous Microprocessors (1989 1999) and 3) Asynchronous CPU Design in the 21st Century.

2.4.1 Early Generation Asynchronous CPU Machines

At a very early stage in computer history, many asynchronous machines were developed. Table 3 shows a summary of the early generation asynchronous CPU machines. The ORDVAC, an early computer built by the University of Illinois in 1952, came to be known as the von Neumann architecture. Its purpose was to perform ballistic trajectory calculations for the US Military [38].

The WEIZAC, one of the world's first fully electronic computers, and the first computer in Israel, was built in 1955 at the Weizmann Institute of Science (WIS). It was an asynchronous computer [39]. The ILLIAC II was the first completely asynchronous, speed independent computer design ever built. It was constructed in 1962 and was the most powerful computer at the time [38], [40]. Atlas was one of the world's first supercomputers built in 1962 from the University of Manchester. Atlas did not use a synchronous clocking mechanism and could therefore be considered to be an asynchronous processor [41], [42].

In 1973, the "Flexible Asynchronous Microprocessor" [43] was described by Saab-Scania in Sweden. Unfortunately, the paper is not very comprehensive and so it is impossible to determine which asynchronous design technique was used. Similarly, a related patent from

International Business Machine (IBM) [44] was published in 1984 but contains few explanatory details regarding their asynchronous technique.

In 1987, a patent was granted to Texas Instruments (TI) [45] related to the design of an asynchronous processor. The patent documents show details of their handshaking technique, which uses pulsed completion signals implemented using Domino circuits.

Table 3: 1st Generation Asynchronous CPU Machines

Year	Source	Description
1952	University of Illinois [38]	ORDVAC, von Neumann architecture
1955	Weizmann Institute of Science (WIS, Israel) [39]	WEIZAC, one of the world's first fully electronic computers
1962	University of Illinois [38], [40]	ILLIAC II,the first completely asynchronous, speed independent computer
1962	University of Manchester [41], [42]	Atlas, one of the world's first supercomputers
1973	Saab-Scania (Sweden) [43]	Flexible Asynchronous Microprocessor
1984	International Business Machine (IBM) [44]	Weak synchronization and scheduling among concurrent asynchronous processors
1987	Texas Instruments (TI) [45]	Asynchronous high speed processor having high speed memories with domino circuits contained therein

2.4.2 Asynchronous Microprocessors

From 1989, after the California Institute of Technology group (under Prof. Alain Martin) announced their 16-Bit RISC-like asynchronous microprocessor using Delay Insensitive technology [46], [47], it can be said that asynchronous CPU design based on single-chip microprocessors started to take off. This work set the basis for a number of CPU explorations during the following decade and beyond.

Table 4, below, is a summary of this 2nd Generation Asynchronous Microprocessors design activity.

Table 4: 2nd Generation Asynchronous Microprocessors

Year	Source	Description
1989	Caltech [46], [47]	16-bit RISC-like asynchronous microprocessor. Four-phase handshaking, dual-rail encoded Delay Insensitive (DI)
1989	Mitsubishi Electric, Sharp and Osaka University [48]	Self-timed clockless Data-Driven Microprocessor
1990	U. California, Berkeley [49]	DSP - four-cycle handshake protocol using DCVSL logic
1990	Utah University, Israel Institute of Technology [50]	A3000 processor, asynchronous version of a pipelined R3000
1993	Israel Institute of Technology [51]	ST-RISC (Self-Timed Reduced Instruction Set Computer) processor
1992	RIST, Korea [52]	Fully Asynchronous Microprocessor (FAM), uses DCVSL circuits
1992 1993	Manchester University [53]–[55]	self-timed ARM microprocessor uses bundled-data, bounded delay model
1992	Stanford University [56]	dynamic clocking - self-timed pipeline-sequencing method
1993	Genoa University [57], [58]	Asynchronous RISC micro-controller based on SGS-Thomson ST9 (ST9026 model)
1993	Tokyo Institute of Technology [59]	TITAC (Tokyo Institute of Technology Asynchronous Chip) CPU uses Dual-Rail 2-phase data transfer
1993	Utah University [60]	NSR (Non-Synchronous RISC) Processor is a pipelined and decoupled 16-bit processor with only sixteen instruction.

1994	Manchester University [61]	AMULET1 processor is 32-bit RISC microprocessor which used a two-phase bundled data design style based closely on Sutherland's Micropipelines.
1994	Caltech [62]	100MIPS GaAs Asynchronous Microprocessor
1995	University of Adelaide [63]	ECSTAC (Event Controlled Systems Temporally-Specified Asynchronous CPU), a RISC style 8-bit Microprocessor.
1996	Utah University [64]	A self-timed decoupled, pipelined architecture based on Motorola 88100 instruction set.
1996	Manchester University [65]	SCALP Superscalar Asynchronous Low-Power Processor
1997	Hong Kong Polytechnic University [66]	"ASYNMPU" processor - 8/16bit CISC type Asynchronous Processor
1997	Newcastle University [67]	Asynchronous Processor design using Petri Nets using two-phase Micro-pipelined delay assumption with C-elements
1997	Caltech [2]	Asynchronous MIPS R3000 Processor, QDI, Four-Phase handshaking protocol and Dual-Rail encoding: HB (Half-Buffer), PCHB (Pre-Charge-Logic Half Buffer) and PCFB (Pre-Charge-Logic Full Buffer).
1997	Manchester University [68], [69]	AMULET-2e processor adopted a four-phase bundled data design style and includes 4Kbyte pipelined cache and a flexible memory interface along with assorted programmable control functions.
1998	Manchester University [70]	AMULET3, high-Performance Self-Timed ARM Microprocessor supporting ARM version-4T and 16-bit Thumb instruction set
1998	Tokyo Institute of Technology and University of Tokyo [71], [72]	"TITAC-2: An asynchronous 32-bit microprocessor based on Scalable-Delay-Insensitive model, 32-bit processor based on MIPS R2000 based instruction set
1998	Israel Institute of Technology [73]	Kin, High Performance Asynchronous Processor Architecture
1998	Eindhoven University of Technology and Philips Research Laboratories [74]	An Asynchronous Low-Power 80C51 Microcontroller. Used Tangram VLSI-programming language and tool-set to compile the design automatically to a standard-cell netlist
1998	France Telecom – CNET Grenoble [75]	ASPRO-216, a Standard-Cell QDI 16-Bit RISC Asynchronous Microprocessor described in a high level sequential CHP program
1998	Technical University of Denmark and LSI Logic [76]	Asynchronous TinyRISC TR4101 Microprocessor core, an asynchronous version of the TR4104

The RISC machine of [46], [47] was based on a four-phase handshaking protocol as well as dual-rail encoded Delay Insensitive (DI) circuits with C-elements for their Completion Tree. Two integrated circuit versions have been fabricated to date: one in 2μm MOSIS SCMOS, and another in 1.6μm MOSIS SCMOS. The performance results were 12 MIPS for a 2μm version and 18 MIPS for a 1.6μm version. The chips were entirely Delay Insensitive except that they required isochronic forks.

The Self-timed Data-Driven Microprocessor was introduced by Mitsubishi Electric, Sharp and Osaka University [48] in 1989. This work used Self-Timed circuits free of any system clocks. Delay elements were purposely added to the SEND signal line to guarantee an appropriate data-processing time through the logic circuit between the latches. The following year a micro-processor based DSP was described by UC-Berkeley group [49]. The architecture is based on a four-cycle handshake protocol and was implemented using DCVSL logic with C-elements for handshaking control.

The A3000 processor from University of Utah and Israel Institute of Technology was also published in 1990 [50]. The A3000 is an asynchronous version of a pipelined R3000, a 32-bit microprocessor developed by MIPS in 1988 and DLX which is simplified version of the MIPS R3000 processor. The A3000 also employed dual-rail encoding and completion detection logic

using C-element just as for the processor in [46]. Later, in 1993, the Israel Institute of Technology group described the ST-RISC (Self-Timed Reduced Instruction Set Computer) processor [51].

The Fully Asynchronous Microprocessor (FAM) developed by RIST, Korea [52] employed DCVSL circuits for its completion signals and combinational logic. The FAM achieved 300MIPS with 0.5μm CMOS technology. It uses 71,000 transistors occupying an area of 6400μm x 4900μm.

The group led by Furber and Garside at the University of Manchester, UK, commenced the development of a self-timed ARM microprocessor in around 1992 [53]–[55]. The methodology applied was based on Sutherland's "Micropipelines", a bundled-data, bounded delay model. The following year, they published the asynchronous version of ARM6 [55].

The STRiP processor was proposed by Stanford University [56]. The key concept in STRiP was a self-timed pipeline-sequencing method called "dynamic clocking". The Strip machine is quite different to other asynchronous architectures in that it is essentially a synchronous processor with an adjustable clock. The clock period is determined using C-elements to approximate the worse-case critical path thereby tracking the instantaneous performance of the machine. The current clock period is set by the slowest critical path. The architecture would appear to be very sensitive to PVT variability as every pipeline stage has to be "tuned" and optimised to achieve optimal performance and the tracking relies on the relative delay behavior of the C-elements and the various CPU blocks.

In 1993, the group from University of Genoa, Italy, proposed an asynchronous RISC micro-controller [57], [58] based on the specification of the SGS-Thomson ST9 (ST9026 model) micro-controller family. The design utilized Delay Insensitive blocks together with conventional (clocked) modules. In the same year, the TITAC (Tokyo Institute of Technology Asynchronous Chip) CPU was presented by the Tokyo Institute of Technology, Japan [59]. This processor used a dual-rail 2-phase data transfer scheme. Erik Brunvand (University of Utah) published details of the NSR (Non-Synchronous RISC) Processor in 1993 [60]. NSR is a pipelined and decoupled 16-bit processor with only sixteen instructions. The prototype of the NSR processor was implemented within seven Actel FPGAs, with each of the pipeline stages using one or two of the FPGA chips. The self-timed blocks in the processor communicate over Bundled Data channels in the style of micro-pipelines.

In 1994, the first silicon of the AMULET1 processor was fabricated, designed between 1991 and 1993 by researchers at the University of Manchester [61]. The AMULET1 is a 32-bit RISC microprocessor which used a two-phase bundled data design style based closely on Sutherland's Micropipelines. Also, the group at the California Institute of Technology [47] published details of a 100MIPS GaAs Asynchronous Microprocessor [62]. In 1995, a RISC style 8-bit Microprocessor ECSTAC (Event Controlled Systems Temporally-Specified Asynchronous CPU) was proposed by the University of Adelaide, Australia [63]. The ESCTAC used Fundamental Mode (FM) free-flow pipelines which is faster and smaller than normal Delay Insensitive design. The technology is similar to the Bundled Data (BD) model with the back-propagating acknowledge (ack) signals removed [77]. In 1996, the University of Utah introduced the "Fred Processor" which has a self-timed decoupled, pipelined architecture based roughly on the NSR and also using micro-pipelines. The Fred instruction set is taken directly from the Motorola 88100 set [64].

In [65] (1996), Endecott from the Manchester group described details of the SCALP superscalar asynchronous processor. Solutions were proposed to the critical problems of checking for dependencies between instructions in the asynchronous superscalar architecture, and ensuring that the instructions were issued in their correct order.

In 1997, The Hong Kong Polytechnic University announced "ASYNMPU" processor which is an 8/16bit CISC type Asynchronous Processor that used Sutherland's Micro-pipelines for their asynchronous implementation [66]. The University of Newcastle, England, published an asynchronous processor design based on Petri Nets with two-phase Micro-pipelined delay assumptions and C-elements [67]. In the same year, the Caltech group announced Asynchronous MIPS R3000 Processor [2] using 0.6μm MOSIS SCMOS technology and 280 MIPS performance expectation and 7W power consumption. They used Quasi Delay-Insensitive (QDI) circuit design, Four-Phase handshaking protocol and Dual-Rail encoding with three types of reshuffling which are HB (Half-Buffer), PCHB (Pre-Charge-Logic Half Buffer) and PCFB (Pre-Charge-Logic Full Buffer). TITAC-2, proposed by Tokyo Institute of Technology and University of Tokyo [71], [72] in 1998 is an asynchronous 32-bit microprocessor based on a novel scalable DI (SDI) model where the previous TITAC-1 [59] was a simple 8-bit processor (based on QDI) that used a MIPS R2000 based instruction set. SDI assumes that the relative delay ratio between any two components is bounded. The performance test results was 52.3 MIPS using the Dhrystone V2.1 benchmark. The AMULET-2e processor introduced by the University of Manchester

[68], [69] adopted a four-phase bundled data design style and includes 4Kbyte pipelined cache and a flexible memory interface along with assorted programmable control functions. The Branch Target Cache unit and Halt unit added many power and performance benefits to the AMULET-2e compared to the previous AMULET1.

In 1998, The Kin high performance asynchronous processor architecture was introduced from Israel Institute of Technology [73]. The "AMULET3: A High-Performance Self-Timed ARM Microprocessor" was published in 1998 from University of Manchester [70]. This is the third generation asynchronous ARM processor from the group, and supports ARM version-4T and 16-bit Thumb instruction set. The AMULET3 achieved 100MIPS (measured with Dhrystone 2.1) on a 3.5μm process. The subsystem of AMULET3 is connected through the MARBLE on-chip bus which is similar in concept to ARM's AMBA bus. AMULET3 included the enhanced version of the Branch Prediction mechanism used on AMULET2e. AMULET3 adopts a reorder buffer to increase pipeline performance and has enhanced Branch Prediction and Halting circuits compared to AMULET2e.

Details of an asynchronous 80C51 microcontroller were published by Eindhoven University of Technology and Philips Research Laboratories [74] in 1998. The chip was designed in 0.5μm CMOS process and shows a power advantage of a factor of four compared to a synchronous implementation. They have used Tangram VLSI-programming language and toolset to compile the design automatically to a standard-cell netlist. In the same year, the QDI ASPRO-216 architecture was described by researchers at France Telecom – CNET Grenoble [75]. The design flow and circuit style used for ASPRO were based directly on Martin's original methodology [47]. As the first processor described by a high level sequential CHP program, ASPRO was based on a unique instruction set architecture that was set up to take advantage of asynchronous design styles. The performance was 200 peak MIPS at 0.5 Watt using a 0.25μm technology. The asynchronous TinyRISC TR4101 Microprocessor core, developed by the Technical University of Denmark and LSI Logic [76], is an asynchronous version of the TR4104. It implements the 32-Bit MIPS-II instruction set architecture and the MIPS16 application specific extension 16-Bit compressed instruction set from LSI Logic called ARISC. The ARISC design uses four-phase handshaking and it performs from 74 to 123 MIPS depending on the instruction mix in 0.35μm CMOS process with 635 MIPS/Watt power efficiency. Balsa was introduced in 1999 from the Manchester group [14].

2.4.3 Asynchronous CPU Design in the 21st Century

Recently, commercial industry seems to have become interested in asynchronous CPU design again, almost certainly due to the extreme low-energy requirements of "always on" Internet of Things applications. As a result, many companies are actively developing asynchronous CPUs for commercial purposes, including Handshake Solution, Tiempo, Theseus Logic, Wave Computing, ETA Compute and Fulcrum (Intel). In addition, many new asynchronous design tools, which have been highlighted in Section 2.1.4, are beginning to emerge out of those commercial activities. Table 5 shows a summary of this 3rd Generation Asynchronous CPU design activity. It is very likely that there are other activities currently in commercial "stealth" mode or otherwise yet to be published.

Table 5: 3rd Generation Asynchronous CPU Design in the 21st Century

Year	Source	Description
2000	University of Manchester [78]	AMULET3i, third generation asynchronous ARM compatible microprocessor, MARBLE asynchronous on-chip bus
2002	Chungbuk National University ,Korea [79], [80]	Lower power A8051, DI delay model, 4-phase handshake protocol
2002	Manchester University [81]	SPA, synthesisable, self-timed, ARM-compatible processor
2003	ETRI ,Korea [82]	ALTHEA, 32-Bit processor, Extended Instruction Set Computer (EISC) ISA
2003	Caltech [83]	Lutonium processor, sub-nanojoule asynchronous 8051 micro-controller using QDI
2003	Cornell University [84]	SNAP (Sensor-Network Asynchronous Processor), 16-bit message passing asynchronous processor, QDI
2005	Handshake Solutions [85]	HT80C51, Low power, asynchronous 80C51 implementation
2006	Handshake Solutions [86]	ARM996HS, 32-bit RISC CPU core ARMv5TE
2007	Fulcrum Microsystems [87]	Vortex, 32-bit integer datapath and executes up to 9 instructions per cycle, QDI
2008	Chungbuk National University ,Korea [88]	AEM32, 32-Bit asynchronous ARM9 processor, 2.6x higher performance than AMULET3i
2008	University of Arkansas [89], [90]	8051-compliant micro-controller intended for extreme environments, NCL
2008	Tiempo [91]	TAM16, 16-Bit microcontroller, QDI
2009	ETRI ,Korea [92]	Asynchronous MSP430 microprocessor
2015	Caltech [93]	Inherently radiation-hard 8-Bit AVR microcontroller, QDI
2015	USC/PUCRS [94]	Blade – A Timing Violation Resilient Asynchronous Template, BD
2015	IBM/Cornell [95]	TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip, QDI
2017	Eta Compute [12]	EtaCore, ARM Cortex-M3 processor compatible asynchronous CPU
2018	Intel [96]	Loihi: A Neuromorphic Manycore Processor with On-Chip Learning

In 2000, AMULET3i was introduced and this was the third generation asynchronous ARM compatible microprocessor subsystem developed at the University of Manchester. It is internally modular, being based around the MARBLE asynchronous on-chip bus, and is also extensible through the addition of conventional clocked synthesizable peripherals via an on-chip synchronous peripheral bus [78].

In 2002, a low power A8051 was designed by Chungbuk National University in Korea [79], [80]. The group used a DI (Delay Insensitive) delay model for its asynchronous operation with a 4-phase handshake protocol for the data transmission and dual-rail encoding.

The Manchester University group introduced SPA, a synthesisable, self-timed, ARM-compatible processor core, in 2002. The SPA core uses the Balsa synthesis system to derive its dual-rail logic. Comparing the synthesised secure dual-rail implementation with the synthesised bundled-data version shows an overall cost factor of 3.6× in terms of transistor count [81].

In 2003, ALTHEA 32-Bit processor was introduced from the ETRI group in Korea [82]. This group used the Extended Instruction Set Computer (EISC) ISA [97] for this CPU core. The core uses Bundled-Data technology and used Handshake Solution's Haste and TiDE Flow. This chip focuses on achieving an ultra-low power micro-architecture.

The Lutonium processor is a sub-nanojoule asynchronous 8051 micro-controller using QDI technology proposed by Martin's group at Caltech. The core targets extremely low power systems and implements a deep-sleep mechanism [83].

The Sensor-Network Asynchronous Processor (SNAP) proposed by the Cornell University group, is based on a 16-bit message passing asynchronous processor. SNAP is composed of quasi-delay-insensitive(QDI) asynchronous circuits [84].

Handshake Solution developed the "HT80C51" in 2005 [85]. Following this, the company designed an ARM996HS architecture in 2006. The ARM996HS chip has 32-bit RISC CPU core ARMv5TE architecture using their HASTE design language [86].

In 2007, the "Vortex", a superscalar asynchronous processor was introduced by Fulcrum Microsystems. Vortex CPU supports a 32-bit integer datapath and executes up to 9 instructions per cycle. The low-level asynchronous circuitry was based on the integrated pipelining templates (WCHB, PCHB, PCFB) and a QDI timing model. Layout was performed entirely by hand in Magic [87].

The AEM32 32-bit asynchronous ARM9 processor was proposed in 2008 by Chungbuk National Unversity in Korea [88]. The group achieved 2.6x higher performance than AMULET3i. An asynchronous 8051-compliant micro-controller intended for extreme environments was described by researchers at the University of Arkansas in 2008 [89], [90]. The 8051

design used Null Convention Logic technology. Tiempo developed the asynchronous TAM16 Core [91] in 2008 which was a 16-Bit microcontroller with their own power-efficient instruction set. Tiempo used QDI technology for the CPU design. In 2009, Asynchronous MSP430 microprocessor was introduced from the ETRI group in Korea [92]. MSP430 is the ISA from Texas Instrument [98]. The group used the Balsa design flow.

In 2015, the Caltech group introduced DD1 [93] which is an inherently radiation-hard 8-Bit AVR microcontroller implementation. Using a near-threshold quasi-delay-insensitive (QDI) methodology, DD1 employed PCHB asynchronous logic and contains full-custom radiation-hard memories and logic cells. The resulting microcontroller achieved a power figure of 18μW/MIPS in 40nm Bulk CMOS. The USC and PUCRS group introduced an asynchronous bundled-data resilient template called Blade [94]. They designed a 3-stage OpenCore MIPS CPU called Plasma and the CPU has 19% performance boost over the synchronous design and a 30-40% improvement when synchronous PVT margins are considered. IBM and Cornell group introduced TrueNorth chip design which has 65mW 1 Million Neuron Programmable Neurosynaptic architecture [95]. The design used mixed synchronous-asynchronous approach. They used QDI asynchronous design style for the communication and control circuits.

In 2017, Eta Compute announced the ARM Cortex-M3 processor compatible asynchronous CPU called "EtaCore" using their DIAL technology [12]. Their CPU core operates at 0.25V on 90nm technology. More recently, Intel announced "Loihi: A Neuromorphic Manycore Processor with On-Chip Learning" using asynchronous technology in 2018. Loihi uses Bundled-Data technology [96].

Although numerous asynchronous CPUs had been developed to this stage, and there are number of advantages in using NCL for CPU design, very few NCL versions of CPU architectures have been developed. It is difficult to say why this is the case. It may have been simply that the technology was patented until around 2014–15 so there was a reticence in the research community to use it, while the commercial community was reluctant to pay royalties for its commercial use. This situation may change now that the basic patents are expiring and NCL is starting to move into the public domain.

2.5 RISC-V Background and Asynchronous RISC-V Design

In the previous asynchronous CPU history, many institutions and companies used their own instruction sets to take advantage of asynchronous technology but some used the

existing ISA such as ARM, MIPS, AVR, MSP430 or 8051 CISC ISA. A key benefit of using an existing ISA are that the existing compiler environments can be reused for the asynchronous CPU design. Further, because they are using a compatible ISA, a company's CPU customers can reuse their application software for the new asynchronous processor with few or no changes. Even some asynchronous CPUs are pin compatible with their synchronous counterparts. This thesis uses the RISC-V ISA which is an open Instruction Set Architecture introduced by University of California, Berkeley in 2011.

2.5.1 What is the RISC-V ISA?

RISC-V [4] is an open, free ISA that was started in 2010 at UC-Berkeley by a group led by Prof. David Patterson. The ultimate intention is to provide a long-lived, open ISA with significant infrastructure support, including documentation, compiler tool chains, operating system ports, reference software simulators, cycle-accurate FPGA emulators, high-performance FPGA computers, efficient ASIC implementations of various target platform designs, configurable processor generators, architecture test suites, and teaching materials [99]. RISC-V ISA is a modular design rather than an incremental ISA and identifies its seven goals as: Cost, Simplicity, Performance, Isolation of architecture from implementation, Room for growth, Program size and Ease of programming, compiling and linking. The key features of the RISC-V ISA, as described by its developers, are [99]:

- it offers an open and extensible software and hardware environment.
- As an open ISA, it aims to deliver easier support from a broad range of operating systems, software vendors and tool developers.
- The open source hardware RISC-V does not rely on a single supplier – it offers multiple suppliers, therefore, supports unlimited potential for future growth.
- No other ISA is structured like the RISC-V ISA, allowing for user extensibility of the architecture without breaking existing extensions or incurring software fragmentation

2.5.2 Advantages of using RISC-V ISA for an Asynchronous CPU Design

The RISC-V is the most modern ISA and those supporting its architecture are pursuing a simplified and extendable architecture and continue to actively update its extension modules.

[4] pronounced "risk-five"

Compared to previous ISAs such as MIPS and ARM, RISC-V is much simpler but has similar code density, fewer condition codes and branch delay slots. A complete description of the ideas guiding the RISC-V architecture can be found in [100].

The RISC-V instruction set has been selected for this work based on the observation that it can lead to simpler execution control and is potentially well suited to asynchronous implementation as it matches more closely the data-flow behavior of an asynchronous CPU. In addition, as an open ISA, RISC-V is supported by a rapidly evolving "eco-system" of tools and design resources, particularly compiler environments from the open source community, along with academic and commercial RTL implementations that will be available for comparison with our asynchronous CPU core.

2.6 Summary

In this chapter, the background of the research has been discussed including previous research relating to asynchronous technology. It has highlighted why asynchronous technology is interesting at the moment and shows various styles of asynchronous technology such as BD, QDI and NCL. The background of Null Convention Logic, the particular asynchronous design technology used for this research, has been discussed, indicating the differences between the NCL and other Asynchronous technologies such as Bundled-Data and QDI.

It is clear that asynchronous techniques have been used for the entire history of computing, from the early generation computer machines to the latest "neuromorphic" processors. Finally, we briefly introduced the RISC-V ISA, the instruction sets that will be implemented in this research.

It has been found that NCL offers a number of advantages to CPU design. Ultimately, the objective here is to develop a CPU core using NCL technology. Before this is possible, it is necessary to develop a dedicated methodology that will result in a straightforward design process. This will be the topic of the next chapter.

Chapter 3

NCL Design Methodology

In this chapter, the detailed methodology used to design NCL circuits is described. Firstly, the NCL compilation, simulation and debug solutions are introduced. An analysis is presented using a number of small designs that compares and contrasts three alternative approaches to the hardware description of NCL circuits. In the subsequent sections, the NCL based ASIC design methodology is discussed from the NCL gate design to the semiconductor chip implementation. Finally, a methodology is presented for the implementation of NCL circuit on commercial FPGA devices and their tools, including the FPGA cell library design methods.

3.1 NCL Design Tools

To design NCL circuits we can use a general Hardware Description Language (HDL) such as Verilog, System Verilog and/or VHDL. This method directly instantiates the NCL gates using a structural gate description. In this way, the description represents a direct translation of the schematic into text. However, this method is very inefficient and can cause debugging problems as well as portability issues. For this reason, a better solution will be to use a dedicated programming language and compiler tools for NCL. This section introduces two such dedicated tools called UNCLE and NELL.

3.1.1 UNCLE

The Unified NCL Environment (UNCLE) is a synthesis tool which converts clock-based behavioral Verilog to an NCL Verilog gate level net-list. UNCLE was developed by Mississippi State University in 2011 and the executable files and manual are available in the public domain (from their web-site). UNCLE has no licensing restrictions or effective limitations for any purpose. It uses general Verilog synthesis tools such as *Design Compiler* from Synopsys or

RTL Compiler from Cadence to generate a clocked Verilog gate-level net-list as an intermediate file. Using these commercial synthesis tools, UNCLE first generates a Verilog gate-level net-list file and then converts that to the NCL net-list file. The tool automatically executes these synthesis steps using Python scripts[1] [15].

As a toolset for creating dual-rail asynchronous designs using NCL, UNCLE supports both data-driven and control-driven (i.e., Balsa-style) styles. The specification level is RTL, which means that the designer is responsible for creating both data-path (registers and compute blocks) and control (finite state machines, sequencers). Designs are specified in Verilog RTL, and uses the commercial synthesis tools to synthesize this to a net-list of D-flip-flops, latches, combinational logic, as well as special gates known by the toolset. The Uncle toolset performs single-rail to dual-rail conversion and then generates the acknowledge network to make the NCL net-list *live* and *safe*. The resulting gate level net-list can then be simulated within a standard Verilog simulator or serve as the input to an ASIC environment for transistor level simulation. Performance optimization is also supported via latch movement to balance data/acknowledge delays. An internal simulator is included in the UNCLE suite that reports gate orphans/cycle time and includes NLDM (Non-Linear Delay Model) timing. The toolset has a regression test suite that includes several examples from both supported design styles [15].

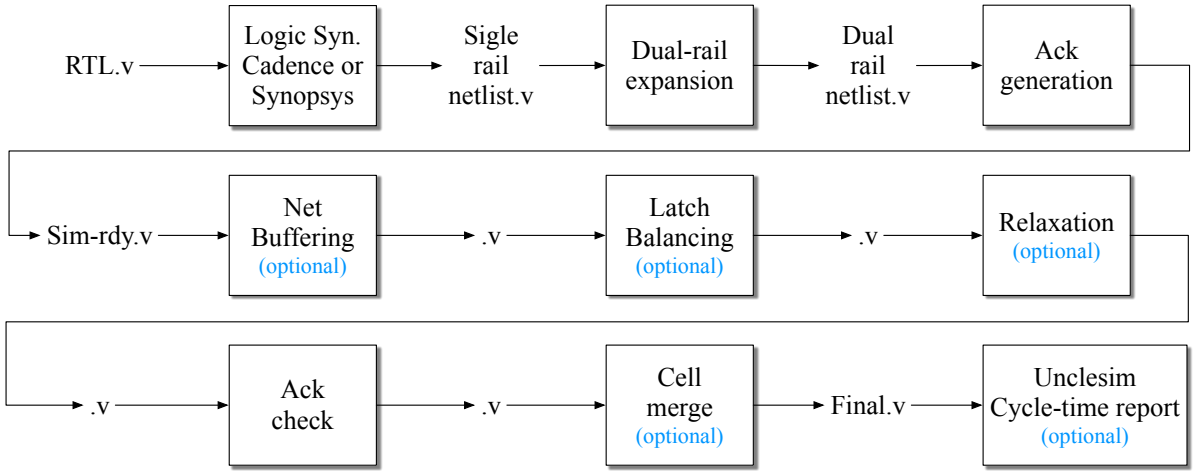

Figure 15: UNCLE Synthesis Flow - redrawn from [15]

[1]UNCLE is not a commercial tool and to make the conversion work easier, I wrote an automated script using Python on top of UNCLE tool. Appendix-A "UNCLE Project Generation Manual" has the User Guide and this Python code.

Figure 15 shows the Synthesis Flow Diagram of UNCLE. As already mentioned, the flow is controlled by Python scripts that invoke the various tools in turn. The input RTL level behavioral Verilog design is first converted to a conventional single rail net-list using the Logic Synthesis tools from Cadence (RTL Compiler) or Synopsys (Design Compiler). The single-rail netlist from the commercial synthesis tools is then expanded to a dual-rail netlist using the dual-rail expansion tools. The gates and registers are further expanded to their actual dual-rail implementations. The *ack* network is then generated, at which point the gate level netlist is simulation-ready. The steps after this point are optional optimizations and checking. The ack checker is a tool that reverse-engineers the ack network to mechanically check its correctness. This is primarily included as a check for coding errors in the ack generation tool when new approaches to the ack network generation module are tested.

UNCLE has four types of optimization flows: Net Buffering, Latch Balancing, Relaxation and Cell Merging.

Net buffering is a performance optimization step that reads an external timing data file for the target library, where the timing data is represented in terms of NLDM lookup tables for output transition time and propagation delay based on input transition time and output capacitive load (the timing data file also contains pin capacitance information).

Latch balancing is a performance optimization for the data-driven style that moves half-latches in the netlist to balance data delays with ack delays. In a linear, multi-stage pipeline, the stage with the longest delay loop formed by the forward data path and the backwards ack path sets the pipeline's maximum throughput. In data-driven finite state machines, the longest loop delay is formed by the delay through the combinational logic plus the backwards delay path of the ack network. The ack delay is dependent on the number of destination points, which in turn sets the completion network depth, while the data delay depends on the data logic complexity.

Relaxation is an optimization that searches for redundant paths between a set of primary inputs and a primary output in a combinational netlist. 'Eager' gates that have reduced transistor counts are placed on the redundant paths, with all primary inputs having at least one path to the primary output that go through non-eager (i.e., input-complete) gates. Uncle implements area-driven relaxation.

A final *Cell Merging* is performed in which adjacent gates with no fan-out are merged into more complex gates. This cell merger is a simpler version of the technology mapper/merging implemented in the ATN (Asynchronous Threshold Networks) tool by Nowick/Jeong from Columbia University [101].

UNCLE finally generates an NCL net-list using its standard NCL component library. By this stage, the net-list can be simulated and is fully implementable on a chip. Using a limited number of Boolean Gates, the tool executes the Synthesis Flow (Design Compiler/RTL Compiler) and replaces the Boolean gates with their equivalent NCL gate combinations. For example, flip-flops will be replaced by dual-rail NCL registers.

3.1.2 NELL

The NCL Equation Logic Language (NELL) was developed by Wave Computing [4] and is an independent NCL description language (plus compiler, simulator and debugger) that does not rely on a standard hardware description language such as VHDL or Verilog [2]. NELL is designed to provide an effective means to express NCL circuits. The NELL language was used for this work because it has a number of specific advantages:

- It uses a C and Verilog-like syntax;
- The NELL environment integrates Compiler, NELL simulator and debugger;
- Generates a System Verilog or Verilog net-list (i.e., .n file to .sv or .v file);
- Natively understands the characteristics of NCL gates rather than Boolean gates;
- Supports race detection algorithms;
- Supports *orphan* management algorithms;
- The design language is specifically optimized design for NCL.

NELL generates a System Verilog or (optionally) a Verilog netlist using a built-in NCL Cell Library. The netlist can then be simulated using a conventional Verilog simulator such as Modelsim, NC-Verilog or VCS with a System Verilog test-bench and Verilog simulation model of NCL gates. NELL is made up of three sub-programs that share much code and many data structures: a *Compiler* that compiles NELL into System Verilog, a *Simulator* that can read test-bench files (NELL test-benches but not System Verilog test-benches) with stimuli input data

[2] We were able to access a NELL executable file and its example designs for this work under Non-Disclosure Agreement with Wave Computing

and verify that the design behaves correctly and, lastly, an *Interactive Debugger* that allows the programmer to investigate the behaviour of the design at the gate level.

Designs that are expressed in NELL could also be expressed in Verilog. The key advantages of using NELL are:

- **Operators**: The "and", " | " of NELL are NCL operations, not Boolean operators. Further, De Morgan's laws do not apply and cannot be used. Thus, even if a standard Boolean HDL was employed, it would nonetheless require extensive modifications to the compilation process to disable the "optimizations" that would break NCL.
- **Variables**: In NCL, variables can have a value or be empty. No other language supports the notion of empty variables.
- **One-hot and Constants**: Using n-rail one-hot control variables is an effective, even essential, NCL design technique. In particular, most other languages would not allow you to assign and test for constant values of 1-hot variables in a natural way.
- **Initialization**: NCL initialization is one of its more important functions. NELL has many tests to ensure that variables are correctly initialized and to catch latent race conditions.
- **Races**: There appears to be a strong connection between completeness and race conditions in that race conditions often arise from violations of completeness. NELL incorporates the notion of completeness, and includes algorithms for race detection.
- **Cycles**: Ultimately, good large scale and efficient NCL design hinges on the management of cycles. The capacity to check for cycle correctness and even for automatic construction of cycles in simple cases (e.g., pipelines) is built into NELL.
- **Orphans**: It is important for good orphan management that orphans not pass through gates. Other languages are clueless about this requirement, and their gate optimizations are likely to create problems with orphans and even to destroy completeness. Again, there appears to be a deep connection between orphan management and NCL completeness that is incorporated into NELL.

At this stage in its development, NELL has very limited automatic optimization options and mostly requires manual optimization by the designer. Thus, it is difficult to directly compare the synthesis results of complex systems such as an asynchronous CPU design using the NELL language and compiler against that from conventional synchronous design tools. On the other hand, synchronous design tools such as from Synopsys and Cadence typically include many

automatic optimization options and also support powerful IP-core designs. However, as NELL is designed specifically to match the characteristics of NCL, it will generate significantly better net-list structures than UNCLE, which is a simple translation program. In addition, NELL offers many more manual optimization opportunities to the circuit designer that come closer to those achieved in a conventional synchronous design flow.

3.2 Tool Flow Analysis

This section compares the results of three possible approaches to the design of complex NCL circuits: Structural-Verilog, UNCLE and NELL. A small number of designs based on NCL theory using Data-Flow (Full Adder with Input/Output Registers) and Control-Flow (Simple State-Machine) models are implemented on an Altera/Intel Cyclone-IV device. The logic resource usage is compared between the three approaches based on these examples. The objective here is to assist in the decision of which style is most suited to the synthesis of complex NCL based applications such as a CPU.

3.2.1 One Bit Full Adder Design

Figure 16 shows a 1-bit Full Adder circuit diagram with input and output registers which can be easily and directly converted to Structural-Verilog (Initialization is not shown in this diagram).

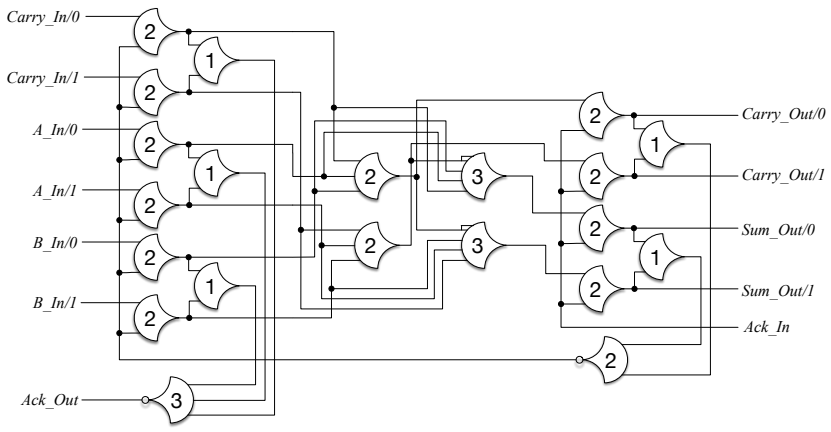

Figure 16: 1-Bit Full Adder with Input and Output Registers

One Bit Full Adder design using Structural-Verilog

Figure 17 has the Structural Verilog design for the 1-Bit Full Adder and it has instantiated five dual_rail_reg sub-modules for the input/output registers and 1-Bit full-adder sub-module.

```
// Input Registers
dual_rail_reg dual_rail_reg_1(reset, carry_in, ack_int, carry_in_int, complete_in[0]);
dual_rail_reg dual_rail_reg_2(reset, a_in, ack_int, a_in_int, complete_in[1]);
dual_rail_reg dual_rail_reg_3(reset, b_in, ack_int, b_in_int, complete_in[2]);

// Full Adder combinational logic
full_adder_comb full_adder_comb_1(a_in_int, b_in_int, carry_in_int, sum_out_int, carry_out_int);

// Output Registers
dual_rail_reg dual_rail_reg_4(reset, carry_out_int, ack_in, carry_out, complete_out[0]);
dual_rail_reg dual_rail_reg_5(reset, sum_out_int, ack_in, sum_out, complete_out[1]);

// Acknowledge generation
th33b th33b_1(ack_out, complete_in[0], complete_in[1], complete_in[2]);
th22b th22b_1(ack_int, complete_out[0], complete_out[1]);
```

Figure 17: 1-Bit Full Adder using Structural Verilog

One Bit Full Adder design using UNCLE

Figure 18 presents a net-list resulting from an UNCLE compilation. UNCLE transforms flip-flops within the synchronous design to the equivalent three-stage dual-rail registers. One flip-flop is replaced by two null–initialized (00) dual-rail latches and one zero-initialized (01) dual-rail latch. Because of this direct replacement, the UNCLE net-list is less efficient compared to the Structural-Verilog or NELL net-list. As the UNCLE net-list has a data initialization value internally, its simulation results are slightly different from those of Structural-Verilog or NELL.

```
th23  g2_U_U_2 (.a ( f_a_d ) , .b ( f_b_d ) , .c ( f_carry_in_d ) , .y ( f_n_15 ));
th23  g2_U_U_1 (.a ( t_a_d ) , .b ( t_b_d ) , .c ( t_carry_in_d ) , .y ( t_n_15 ));
th34w2  g2_U_U_0 (.a ( t_n_15 ) , .b ( f_a_d ) , .c ( f_b_d ) , .d ( f_carry_in_d ) , .y ( f_n_14 ));
th34w2  g2_U_U (.a ( f_n_15 ) , .b ( t_a_d ) , .c ( t_b_d ) , .d ( t_carry_in_d ) , .y ( t_n_14 ));
th33  cgate1 (.a ( acknet1 ) , .b ( acknet0 ) , .c ( acknet2 ) , .y ( acknet6 ));
th22  cgate0 (.a ( acknet3 ) , .b ( acknet4 ) , .y ( acknet5 ));
drlatn  carry_in_d_reg_0_U (.rsb ( reset ) , .ackout ( n2_N ) , .ackin ( acknet5 ) , .f_d ( f_q2_N ) , .t_d ( t_
drlatr  carry_in_d_reg_0_U_0 (.rsb ( reset ) , .ackout ( n1_N ) , .ackin ( n2_N ) , .f_d ( f_q1_N ) , .t_d ( t_
drlatn  carry_in_d_reg_0_U_1 (.rsb ( reset ) , .ackout ( acknet0 ) , .ackin ( n1_N ) , .f_d ( f_carry_in ) , .t_
drlatn  b_d_reg_0_U (.rsb ( reset ) , .ackout ( n2_N_0 ) , .ackin ( acknet5 ) , .f_d ( f_q2_N_0 ) , .t_d ( t_q2_
drlatr  b_d_reg_0_U_0 (.rsb ( reset ) , .ackout ( n1_N_0 ) , .ackin ( n2_N_0 ) , .f_d ( f_q1_N_0 ) , .t_d ( t_q
drlatn  b_d_reg_0_U_1 (.rsb ( reset ) , .ackout ( acknet1 ) , .ackin ( n1_N_0 ) , .f_d ( f_b ) , .t_d ( t_b ) ,
drlatn  a_d_reg_0_U (.rsb ( reset ) , .ackout ( n2_N_1 ) , .ackin ( acknet5 ) , .f_d ( f_q2_N_1 ) , .t_d ( t_q2_
drlatr  a_d_reg_0_U_0 (.rsb ( reset ) , .ackout ( n1_N_1 ) , .ackin ( n2_N_1 ) , .f_d ( f_q1_N_1 ) , .t_d ( t_q
drlatn  a_d_reg_0_U_1 (.rsb ( reset ) , .ackout ( acknet2 ) , .ackin ( n1_N_1 ) , .f_d ( f_a ) , .t_d ( t_a ) ,
drlatn  carry_out_reg_0_U (.rsb ( reset ) , .ackout ( n2_N_2 ) , .ackin ( ackin ) , .f_d ( f_q2_N_2 ) , .t_d ( 
drlatr  carry_out_reg_0_U_0 (.rsb ( reset ) , .ackout ( n1_N_2 ) , .ackin ( n2_N_2 ) , .f_d ( f_q1_N_2 ) , .t_d
drlatn  carry_out_reg_0_U_1 (.rsb ( reset ) , .ackout ( acknet3 ) , .ackin ( n1_N_2 ) , .f_d ( f_n_15 ) , .t_d 
drlatn  sum_reg_0_U (.rsb ( reset ) , .ackout ( n2_N_3 ) , .ackin ( ackin ) , .f_d ( f_q2_N_3 ) , .t_d ( t_q2_N_
drlatr  sum_reg_0_U_0 (.rsb ( reset ) , .ackout ( n1_N_3 ) , .ackin ( n2_N_3 ) , .f_d ( f_q1_N_3 ) , .t_d ( t_q
drlatn  sum_reg_0_U_1 (.rsb ( reset ) , .ackout ( acknet4 ) , .ackin ( n1_N_3 ) , .f_d ( f_n_14 ) , .t_d ( t_n_
```

Figure 18: 1-Bit Full Adder design using UNCLE

One Bit Full Adder design using NELL

Figure 19 shows the NELL program code and Figure 20 shows the Adder combinational logic sub-module which was designed using dual-rail XOR logic. These two modules were assembled into a 1-bit Full Adder and then converted to a System-Verilog net-list after NELL compilation. Figure 21 describes the resulting net-list. Just as for the Structural-Verilog design, NELL generates one dual-rail register at each of the input and output edges. The combinational adder part of the NELL net-list is not optimized therefore it uses more threshold gates.

```
module full_adder(
    input dual a_in,
    input dual b_in,
    input dual carry_in,
    input rail ack_in,
    output dual sum_out(null),
    output dual carry_out(null),
    output rail ack_out
    );
    rail int_ack(null) = ~((^sum) & (^carry_out));
    dual a(null) = a_in & int_ack;
    dual b(null) = b_in & int_ack;
    dual ci(null) = carry_in & int_ack;
    dual sum;
    dual co;
    sum_out = sum & ack_in;
    carry_out = co & ack_in;
    ack_out = ~((^a) & (^b) & (^ci));

    Add(.co(co), .sum(sum), .a(a), .b(b), .ci(ci));
endmodule
```

Figure 19: 1-Bit Full Adder NELL program code

```
module Add(
    output dual co,
    output dual sum,
    input dual a,
    input dual b,
    input dual ci
    );
    dual t;
    Xor( t, a, b );
    Xor( sum, t, ci );
    co = { a/0 & b/0 | a/0 & ci/0 | b/0 & ci/0,
           a/1 & b/1 | a/1 & ci/1 | b/1 & ci/1 };
endmodule

module Xor(
  output dual x,
  input dual a,
  input dual b );
  x/0 = a/0&b/0 | a/1&b/1;
  x/1 = a/0&b/1 | a/1&b/0;
endmodule
```

Figure 20: Full Adder combinational design NELL code

```
//    Nell_Version_14_05_20_08_16__
TH22N_1x full_adder_0_0( .Z ( sum_out_0_[0] ), .A ( x_3_[0] ), .B ( ack_in_0_ ), .I(reset___) );
TH22N_1x full_adder_0_1( .Z ( sum_out_0_[1] ), .A ( x_3_[1] ), .B ( ack_in_0_ ), .I(reset___) );
TH22N_1x full_adder_0_2( .Z ( carry_out_0_[0] ), .A ( co_1_[0] ), .B ( ack_in_0_ ), .I(reset___) );
TH22N_1x full_adder_0_3( .Z ( carry_out_0_[1] ), .A ( co_1_[1] ), .B ( ack_in_0_ ), .I(reset___) );
T11B_1x full_adder_0_4_COMPINV( .Z ( ack_out_0_ ), .A ( \\6_0_  ) );
WTLT11BNC full_adder_0_5( .Z ( int_ack_0_ ), .A ( \\2_0_  ), .I(reset___) );
TH22N_1x full_adder_0_6_ENBRANK( .Z ( a_0_[0] ), .A ( a_in_0_[0] ), .B ( int_ack_0_ ), .I(reset___) );
TH22N_1x full_adder_0_7_ENBRANK( .Z ( a_0_[1] ), .A ( a_in_0_[1] ), .B ( int_ack_0_ ), .I(reset___) );
TH22N_1x full_adder_0_8_ENBRANK( .Z ( b_0_[0] ), .A ( b_in_0_[0] ), .B ( int_ack_0_ ), .I(reset___) );
TH22N_1x full_adder_0_9_ENBRANK( .Z ( b_0_[1] ), .A ( b_in_0_[1] ), .B ( int_ack_0_ ), .I(reset___) );
TH22N_1x full_adder_0_10_ENBRANK( .Z ( ci_0_[0] ), .A ( carry_in_0_[0] ), .B ( int_ack_0_ ), .I(reset___) );
TH22N_1x full_adder_0_11_ENBRANK( .Z ( ci_0_[1] ), .A ( carry_in_0_[1] ), .B ( int_ack_0_ ), .I(reset___) );
T12_1x full_adder_0_12( .Z ( \\0_0_  ), .A ( x_3_[0] ), .B ( x_3_[1] ) );
T12_1x full_adder_0_13( .Z ( \\1_0_  ), .A ( carry_out_0_[0] ), .B ( carry_out_0_[1] ) );
TH22_1x full_adder_0_14( .Z ( \\2_0_  ), .A ( \\0_0_  ), .B ( \\1_0_  ) );
T12_1x full_adder_0_15( .Z ( \\3_0_  ), .A ( a_0_[0] ), .B ( a_0_[1] ) );
T12_1x full_adder_0_16( .Z ( \\4_0_  ), .A ( b_0_[0] ), .B ( b_0_[1] ) );
T12_1x full_adder_0_17( .Z ( \\5_0_  ), .A ( ci_0_[0] ), .B ( ci_0_[1] ) );
TH33_1x full_adder_0_18( .Z ( \\6_0_  ), .A ( \\3_0_  ), .B ( \\4_0_  ), .C ( \\5_0_  ) );
T13_1x full_adder_Add_1_0( .Z ( co_1_[0] ), .A ( \\7_1_  ), .B ( \\8_1_  ), .C ( \\9_1_  ) );
T13_1x full_adder_Add_1_1( .Z ( co_1_[1] ), .A ( \\10_1_  ), .B ( \\11_1_  ), .C ( \\12_1_  ) );
TH22_1x full_adder_0_19( .Z ( \\7_1_  ), .A ( a_0_[0] ), .B ( b_0_[0] ) );
TH22_1x full_adder_0_20( .Z ( \\8_1_  ), .A ( a_0_[0] ), .B ( ci_0_[0] ) );
TH22_1x full_adder_0_21( .Z ( \\9_1_  ), .A ( b_0_[0] ), .B ( ci_0_[0] ) );
TH22_1x full_adder_0_22( .Z ( \\10_1_  ), .A ( a_0_[1] ), .B ( b_0_[1] ) );
TH22_1x full_adder_0_23( .Z ( \\11_1_  ), .A ( a_0_[1] ), .B ( ci_0_[1] ) );
TH22_1x full_adder_0_24( .Z ( \\12_1_  ), .A ( b_0_[1] ), .B ( ci_0_[1] ) );
T12_1x full_adder_Add_Xor_2_0( .Z ( x_2_[0] ), .A ( \\13_2_  ), .B ( \\14_2_  ) );
T12_1x full_adder_Add_Xor_2_1( .Z ( x_2_[1] ), .A ( \\15_2_  ), .B ( \\16_2_  ) );
TH22_1x full_adder_0_25( .Z ( \\13_2_  ), .A ( a_0_[0] ), .B ( b_0_[0] ) );
TH22_1x full_adder_0_26( .Z ( \\14_2_  ), .A ( a_0_[1] ), .B ( b_0_[1] ) );
TH22_1x full_adder_0_27( .Z ( \\15_2_  ), .A ( a_0_[0] ), .B ( b_0_[1] ) );
TH22_1x full_adder_0_28( .Z ( \\16_2_  ), .A ( a_0_[1] ), .B ( b_0_[0] ) );
T12_1x full_adder_Add_Xor_3_0( .Z ( x_3_[0] ), .A ( \\13_3_  ), .B ( \\14_3_  ) );
T12_1x full_adder_Add_Xor_3_1( .Z ( x_3_[1] ), .A ( \\15_3_  ), .B ( \\16_3_  ) );
TH22_1x full_adder_0_29( .Z ( \\13_3_  ), .A ( x_2_[0] ), .B ( ci_0_[0] ) );
TH22_1x full_adder_0_30( .Z ( \\14_3_  ), .A ( x_2_[1] ), .B ( ci_0_[1] ) );
TH22_1x full_adder_0_31( .Z ( \\15_3_  ), .A ( x_2_[0] ), .B ( ci_0_[1] ) );
TH22_1x full_adder_0_32( .Z ( \\16_3_  ), .A ( x_2_[1] ), .B ( ci_0_[0] ) );
```

Figure 21: 1-Bit Full Adder net-list generated by NELL

One Bit Full Adder Design using NELL with Manual Gate Instantiation

Figure 22 illustrates a simplified Adder module which is designed using two TH23 gates and two TH34W2 gates. NELL supports a threshold gate instantiation function in a similar way to Verilog that is intended to allow a design to be hand-optimized. From the compiler result (Figure 23) it can be seen that the net-list has a smaller gate count compared to Figure 21.

```
module full_adder(
    input dual a_in,
    input dual b_in,
    input dual carry_in,
    input rail ack_in,
    output dual sum_out(null),
    output dual carry_out(null),
    output rail ack_out
    );
    rail int_ack(null) = ~((^sum) & (^carry_out));
    dual a(null) = a_in & int_ack;
    dual b(null) = b_in & int_ack;
    dual ci(null) = carry_in & int_ack;
    dual sum;
    dual co;
    sum_out = sum & ack_in;
    carry_out = co & ack_in;
    ack_out = ~((^a) & (^b) & (^ci));
    Add(.co(co), .sum(sum), .a(a), .b(b), .ci(ci));
endmodule

module Add(
    output dual co,
    output dual sum,
    input dual a,
    input dual b,
    input dual ci
    );
    co/0 = "TH23" ( ci/0, a/0, b/0 );
    co/1 = "TH23" ( ci/1, a/1, b/1 );
    sum/0 = "TH34W2" ( co/1, ci/0, a/0, b/0 );
    sum/1 = "TH34W2" ( co/0, ci/1, a/1, b/1 );
endmodule
```

Figure 22: 1-Bit Full Adder - NELL program code - with Gate Instantiation

```
//    Nell_Version_14_10_10_12_18__
TH22N_1x full_adder_0_0( .Z ( sum_out_0_[0] ), .A ( sum_1_[0] ), .B ( ack_in_0_ ), .I(reset___) );
TH22N_1x full_adder_0_1( .Z ( sum_out_0_[1] ), .A ( sum_1_[1] ), .B ( ack_in_0_ ), .I(reset___) );
TH22N_1x full_adder_0_2( .Z ( carry_out_0_[0] ), .A ( co_1_[0] ), .B ( ack_in_0_ ), .I(reset___) );
TH22N_1x full_adder_0_3( .Z ( carry_out_0_[1] ), .A ( co_1_[1] ), .B ( ack_in_0_ ), .I(reset___) );
T11B_1x full_adder_0_4_COMPINV( .Z ( ack_out_0_ ), .A ( \\6_0_  ) );
WTLT11BNC full_adder_0_5( .Z ( int_ack_0_ ), .A ( \\2_0_  ), .I(reset___) );
TH22N_1x full_adder_0_6_ENBRANK( .Z ( a_0_[0] ), .A ( a_in_0_[0] ), .B ( int_ack_0_ ), .I(reset___) );
TH22N_1x full_adder_0_7_ENBRANK( .Z ( a_0_[1] ), .A ( a_in_0_[1] ), .B ( int_ack_0_ ), .I(reset___) );
TH22N_1x full_adder_0_8_ENBRANK( .Z ( b_0_[0] ), .A ( b_in_0_[0] ), .B ( int_ack_0_ ), .I(reset___) );
TH22N_1x full_adder_0_9_ENBRANK( .Z ( b_0_[1] ), .A ( b_in_0_[1] ), .B ( int_ack_0_ ), .I(reset___) );
TH22N_1x full_adder_0_10_ENBRANK( .Z ( ci_0_[0] ), .A ( carry_in_0_[0] ), .B ( int_ack_0_ ), .I(reset___) );
TH22N_1x full_adder_0_11_ENBRANK( .Z ( ci_0_[1] ), .A ( carry_in_0_[1] ), .B ( int_ack_0_ ), .I(reset___) );
T12_1x full_adder_0_12( .Z ( \\0_0_  ), .A ( sum_1_[0] ), .B ( sum_1_[1] ) );
T12_1x full_adder_0_13( .Z ( \\1_0_  ), .A ( carry_out_0_[0] ), .B ( carry_out_0_[1] ) );
TH22_1x full_adder_0_14( .Z ( \\2_0_  ), .A ( \\0_0_  ), .B ( \\1_0_  ) );
T12_1x full_adder_0_15( .Z ( \\3_0_  ), .A ( a_0_[0] ), .B ( a_0_[1] ) );
T12_1x full_adder_0_16( .Z ( \\4_0_  ), .A ( b_0_[0] ), .B ( b_0_[1] ) );
T12_1x full_adder_0_17( .Z ( \\5_0_  ), .A ( ci_0_[0] ), .B ( ci_0_[1] ) );
TH33_1x full_adder_0_18( .Z ( \\6_0_  ), .A ( \\3_0_  ), .B ( \\4_0_  ), .C ( \\5_0_  ) );
TH23_1x full_adder_Add_1_0( .Z ( co_1_[0] ), .A ( ci_0_[0] ), .B ( a_0_[0] ), .C ( b_0_[0] ) );
TH23_1x full_adder_Add_1_1( .Z ( co_1_[1] ), .A ( ci_0_[1] ), .B ( a_0_[1] ), .C ( b_0_[1] ) );
TH34W2_1x full_adder_Add_1_2( .Z ( sum_1_[0] ), .A ( co_1_[1] ), .B ( ci_0_[0] ), .C ( a_0_[0] ), .D ( b_0_[0] ) );
TH34W2_1x full_adder_Add_1_3( .Z ( sum_1_[1] ), .A ( co_1_[0] ), .B ( ci_0_[1] ), .C ( a_0_[1] ), .D ( b_0_[1] ) );
```

Figure 23: 1-Bit Full Adder net-list generated by NELL - with Gate Instantiation

Functional Simulation

The Full Adder net-lists of Structural-Verilog, UNCLE and NELL are all functionally the same and thus their simulation results are also virtually the same. One very minor difference arises from the fact that the Structural-Verilog and NELL net-list have only NULL initialization values but the UNCLE net-list has three-stage registers NULL (00)-DATA (01)-NULL (00) initialization values for the input and output gates of the register triple. Therefore, the UNCLE net-list exhibits greater latency to reach an expected output result.

3.2.2 NCL based State-Machine Control Logic Design

Simple State Sequencer architecture

Figure 24 shows the NCL based "Simple State Sequencer" which was introduced by Fant [18]. It uses four single-rail ring registers and their completion detection gates and the output combinational logic for responding to the input ACK signal (note that initialization is not shown in this diagram). The design has four single rail control outputs which express the current states of the controller. Smith also described a generic NCL based State-Machine [19], which also uses dual-rail registers to save the current status information. Figure 25 shows the dia-

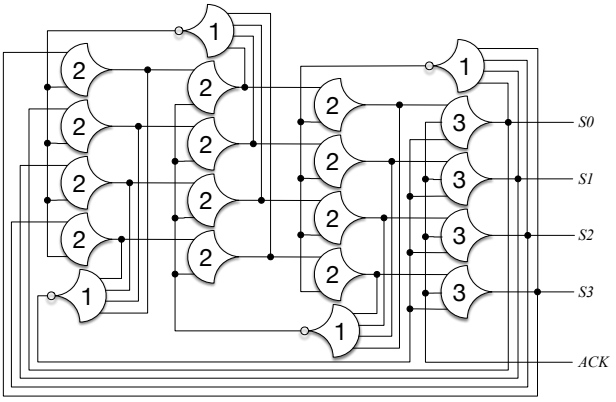

Figure 24: Simple State Sequencer circuit diagram - redrawn from [18]

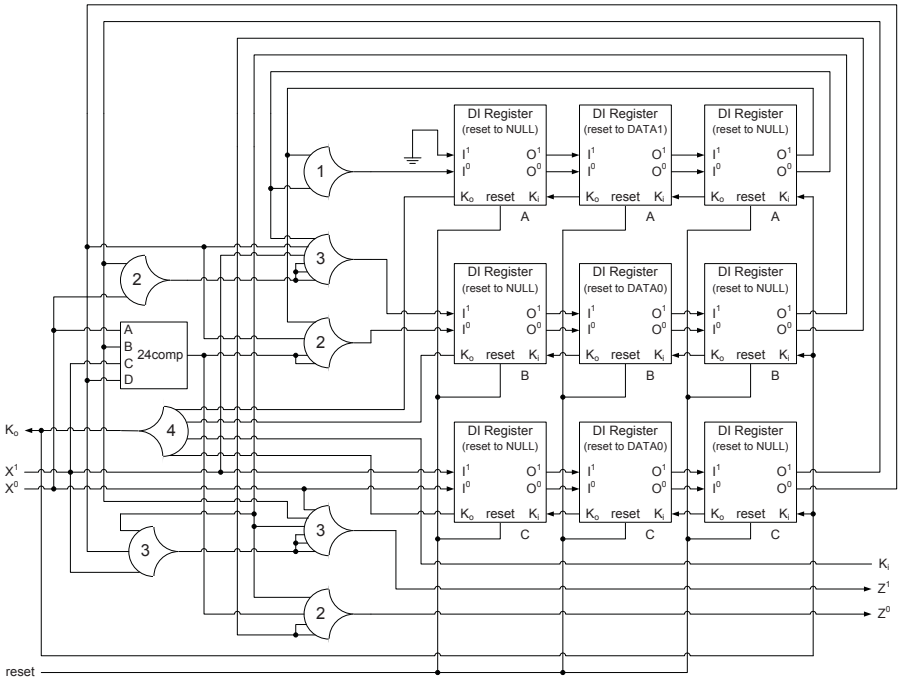

Figure 25: NCL based Dual-Rail State Machine [19]

gram of Smith's dual-rail state machine (for simplicity, drawn with only one dual-rail output). Although the design is not the same as the Simple State Sequencer, it expresses Smith's State-Machine design methodology in a straightforward way.

Simple State Sequencer using Structural-Verilog

The Simple State Sequencer is implemented using Structural-Verilog description (Figure 26).

```
// Module call
reg_1bit_init1 reg_1bit_1(reset, th33_s3 ,th14b_x2 ,th14b_x3 ,th14b_x4 ,th22_11 ,th22_12 ,th22_13);
reg_1bit_init0 reg_1bit_2(reset, th33_s0 ,th14b_x2 ,th14b_x3 ,th14b_x4 ,th22_21 ,th22_22 ,th22_23);
reg_1bit_init0 reg_1bit_3(reset, th33_s1 ,th14b_x2 ,th14b_x3 ,th14b_x4 ,th22_31 ,th22_32 ,th22_33);
reg_1bit_init0 reg_1bit_4(reset, th33_s2 ,th14b_x2 ,th14b_x3 ,th14b_x4 ,th22_41 ,th22_42 ,th22_43);

th14b th14b_1(th14b_x1 ,th22_11 ,th22_21 ,th22_31 ,th22_41);
th14b th14b_2(th14b_x2 ,th22_12 ,th22_22 ,th22_32 ,th22_42);
th14b th14b_3(th14b_x3 ,th22_13 ,th22_23 ,th22_33 ,th22_43);

th33r th33_1(th33_s0 ,th22_13 ,ack_in ,th14b_x1, reset);
th33r th33_2(th33_s1 ,th22_23 ,ack_in ,th14b_x1, reset);
th33r th33_3(th33_s2 ,th22_33 ,ack_in ,th14b_x1, reset);
th33r th33_4(th33_s3 ,th22_43 ,ack_in ,th14b_x1, reset);

th14b th14b_4(th14b_x4 ,th33_s0 ,th33_s1 ,th33_s2 ,th33_s3);
```

Figure 26: State Sequencer with Structural Verilog

Figure 27 is the functional simulation result of the single-rail Simple State Sequencer and the sequence of each state (s0, s1, s2, s3) are changing in order.

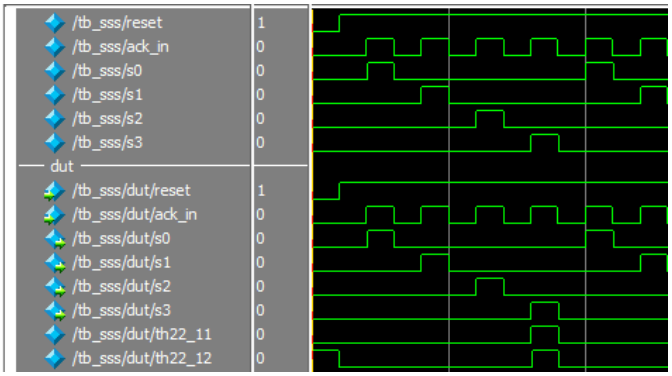

Figure 27: Structural-Verilog based State Sequencer Modelsim Functional Simulation

```
module clk_state_machine(clk, reset, state);
   input clk, reset;
   output [3:0] state;
   reg [3:0] state;
   parameter zero =4'b0001;
   parameter one  =4'b0010;
   parameter two  =4'b0100;
   parameter three=4'b1000;
   always@(posedge clk or negedge reset)
   begin
      if (!reset)
         state = zero;
      else
         case (state)
            zero:  state = one;
            one:   state = two;
            two:   state = three;
            three: state = zero;
         endcase
   end
endmodule
```

Figure 28: State Sequencer clocked design input for UNCLE

Figure 29 is the SignalTap monitoring result on the Altera/Intel Cyclone-IV device. The ACK input signal is generated by the input clock signal and a 20MHz clock is used for this source.

Simple State Sequencer using UNCLE

Figure 28 is the clocked synchronous design of the state-machine, which has been converted

Figure 29: Structural-Verilog based State Sequencer SignalTap Monitoring

to a NCL net-list using UNCLE. Figure 30 gives the resulting net-list showing the synchronous flip-flops converted to NCL dual-rail three-stage registers. The direct conversion from flip-flops to three-stage registers is clearly less efficient compared to the single-stage register designed using TH22 gates.

```
th22  cgate1 (.a ( acknet2 ) , .b ( acknet4 ) , .y ( acknet5 ));
th44  cgate0 (.a ( acknet0 ) , .b ( ackin ) , .c ( acknet3 ) , .d ( acknet1 ) , .y ( acknet4 ));
logic_1  pull1_g (.y ( pull1n ));
thand0  U22_U (.y ( t_n15 ) , .d ( f_state[2] ) , .c ( f_state[3] ) , .b ( t_state[2] ) , .a ( t_state[3] ));
th22  U22_U_0 (.y ( f_n15 ) , .b ( f_state[2] ) , .a ( f_state[3] ));
thand0  U20_U (.y ( t_n16 ) , .d ( t_state[1] ) , .c ( f_state[0] ) , .b ( f_state[1] ) , .a ( t_state[0] ));
th22  U20_U_0 (.y ( f_n16 ) , .b ( t_state[1] ) , .a ( f_state[0] ));
thand0  U19_U (.y ( t_n14 ) , .d ( f_n16 ) , .c ( f_n15 ) , .b ( t_n16 ) , .a ( t_n15 ));
th22  U19_U_0 (.y ( f_n14 ) , .b ( f_n16 ) , .a ( f_n15 ));
thand0  U17_U (.y ( t_n12 ) , .d ( f_state[1] ) , .c ( f_state[3] ) , .b ( t_state[1] ) , .a ( t_state[3] ));
th22  U17_U_0 (.y ( f_n12 ) , .b ( f_state[1] ) , .a ( f_state[3] ));
thand0  U15_U (.y ( t_n13 ) , .d ( t_state[2] ) , .c ( f_state[0] ) , .b ( f_state[2] ) , .a ( t_state[0] ));
th22  U15_U_0 (.y ( f_n13 ) , .b ( t_state[2] ) , .a ( f_state[0] ));
thand0  U14_U (.y ( t_n11 ) , .d ( f_n13 ) , .c ( f_n12 ) , .b ( t_n13 ) , .a ( t_n12 ));
th22  U14_U_0 (.y ( f_n11 ) , .b ( f_n13 ) , .a ( f_n12 ));
thand0  U12_U (.y ( t_n9 ) , .d ( f_state[1] ) , .c ( f_state[2] ) , .b ( t_state[1] ) , .a ( t_state[2] ));
th22  U12_U_0 (.y ( f_n9 ) , .b ( f_state[1] ) , .a ( f_state[2] ));
thand0  U10_U (.y ( t_n10 ) , .d ( t_state[3] ) , .c ( f_state[0] ) , .b ( f_state[3] ) , .a ( t_state[0] ));
th22  U10_U_0 (.y ( f_n10 ) , .b ( t_state[3] ) , .a ( f_state[0] ));
thand0  U9_U (.y ( t_n8 ) , .d ( f_n10 ) , .c ( f_n9 ) , .b ( t_n10 ) , .a ( t_n9 ));
th22  U9_U_0 (.y ( f_n8 ) , .b ( f_n10 ) , .a ( f_n9 ));
thand0  U7_U (.y ( f_n3 ) , .d ( f_state[2] ) , .c ( f_state[3] ) , .b ( t_state[2] ) , .a ( t_state[3] ));
th22  U7_U_0 (.y ( t_n3 ) , .b ( f_state[2] ) , .a ( f_state[3] ));
thand0  U6_U (.y ( f_n4 ) , .d ( t_state[0] ) , .c ( f_state[1] ) , .b ( f_state[0] ) , .a ( t_state[1] ));
th22  U6_U_0 (.y ( t_n4 ) , .b ( t_state[0] ) , .a ( f_state[1] ));
thand0  U5_U (.y ( f_N5 ) , .d ( t_n4 ) , .c ( t_n3 ) , .b ( f_n4 ) , .a ( f_n3 ));
th22  U5_U_0 (.y ( t_N5 ) , .b ( t_n4 ) , .a ( t_n3 ));
thand0  U4_U (.y ( t_n1 ) , .d ( t_n8 ) , .c ( f_N5 ) , .b ( f_n8 ) , .a ( t_N5 ));
th22  U4_U_0 (.y ( f_n1 ) , .b ( t_n8 ) , .a ( f_N5 ));
thand0  U3_U (.y ( t_n2 ) , .d ( t_n14 ) , .c ( t_n11 ) , .b ( f_n14 ) , .a ( f_n11 ));
th22  U3_U_0 (.y ( f_n2 ) , .b ( t_n14 ) , .a ( t_n11 ));
thand0  U2_U (.y ( t_N21 ) , .d ( f_n2 ) , .c ( f_n1 ) , .b ( t_n2 ) , .a ( t_n1 ));
th22  U2_U_0 (.y ( f_N21 ) , .b ( f_n2 ) , .a ( f_n1 ));
drlatn  \state_reg_3__0_U (.rsb ( reset ) , .ackout ( n2_N ) , .ackin ( acknet5 ) , .f_d ( f_q2_N ) , .t_d ( t_q2_N ) , .f_q
drlatr  \state_reg_3__0_U_0 (.rsb ( reset ) , .ackout ( n1_N ) , .ackin ( n2_N ) , .f_d ( f_q1_N ) , .t_d ( t_q1_N ) , .f_q
drlatn  \state_reg_1__0_U (.rsb ( reset ) , .ackout ( n2_N_0 ) , .ackin ( acknet5 ) , .f_d ( f_q2_N_0 ) , .t_d ( t_q2_N_0 )
drlatr  \state_reg_1__0_U_0 (.rsb ( reset ) , .ackout ( n1_N_0 ) , .ackin ( n2_N_0 ) , .f_d ( f_q1_N_0 ) , .t_d ( t_q1_N_0 )
drlatn  \state_reg_2__0_U (.rsb ( reset ) , .ackout ( n2_N_1 ) , .ackin ( acknet5 ) , .f_d ( f_q2_N_1 ) , .t_d ( t_q2_N_1 )
drlatr  \state_reg_2__0_U_0 (.rsb ( reset ) , .ackout ( n1_N_1 ) , .ackin ( n2_N_1 ) , .f_d ( f_q1_N_1 ) , .t_d ( t_q1_N_1 )
drlatn  \state_reg_0__0_U (.rsb ( reset ) , .ackout ( n2_N_2 ) , .ackin ( acknet5 ) , .f_d ( f_q2_N_2 ) , .t_d ( t_q2_N_2 )
```

Figure 30: State Sequencer UNCLE Result (part of Verilog net-list)

Figure 31 displays the functional simulation result of the UNCLE net-list and the output signal of the state-machine is dual-rail (f_state, t_state). The state transition sequence can be easily seen from this simulation result.

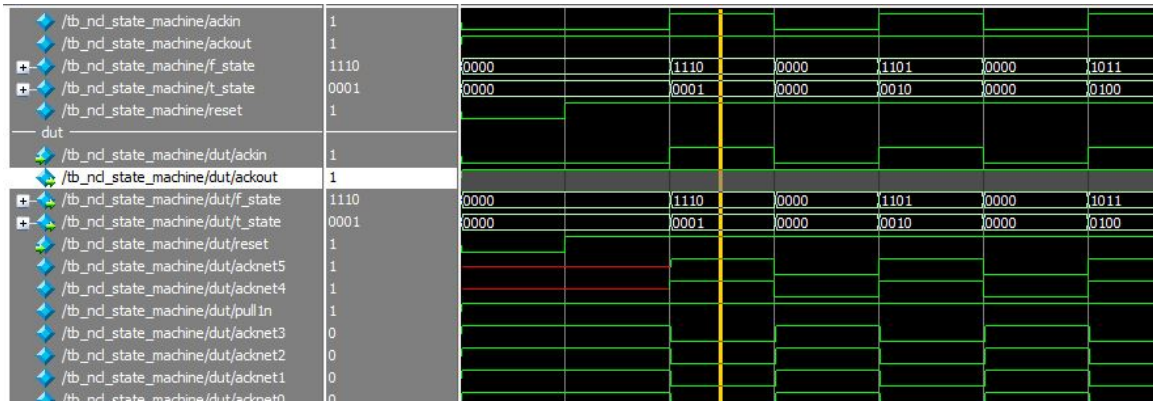

Figure 31: UNCLE Net-list simulation of Dual-Rail State Machine

Figure 32 shows the State-Machine expressed using two-bit Binary output whereas Figure 28 shows the one-hot 4-state State-Machine. Figure 34 shows the dual-rail binary state-machine generated by UNCLE. Again, it can be seen that the net-list uses fewer gates compared to the one-hot encoded machine.

```
module clk_state_machine_binary
   (clk, reset, state);
   input clk, reset;
   output [1:0] state;
   reg [1:0] state;
   parameter zero=  2'b00;
   parameter one=   2'b01;
   parameter two=   2'b10;
   parameter three= 2'b11;
   always@(posedge clk or negedge reset)
   begin
      if (!reset)
        state = zero;
      else
      case (state)
         zero:  state = one;
         one:   state = two;
         two:   state = three;
         three: state = zero;
      endcase
   end
endmodule
```

Figure 32: Binary State-Machine State Sequencer clocked design input for UNCLE

```
parameter n=4;
module state_machine(
   output rail[n] s(null),
   input rail ack_in);
   // rega -> regb -> ns -> s
   rail [n] ns(null);
   rail [n] rega(null);
   rail [n] regb(0);

   rail rega_ack = ~rega; // All Null
   rail regb_ack = ~regb;
   rail ns_ack = ~ns; // 0001 - Init Value
   rail s_ack = ~s;

   rega = {s/3, s/0, s/1, s/2} & regb_ack;
   regb = rega & ns_ack;
   ns = regb & s_ack;
   s= ns & rega_ack & ack_in;
endmodule
```

Figure 33: State Sequencer NELL program

Simple State Sequencer Design using NELL

The state-machine can also be implemented using NELL, as illustrated by the NELL program code in Figure 33, which simply expresses the four-stage state-machine. Figure 35 is the State Sequencer System-Verilog net-list that has been generated by the NELL compiler.

```
th33  cgate0 (.a ( ackin ) , .b ( acknet0 ) , .c ( acknet1 ) , .y ( acknet2 ));
logic_1  pull1_g (.y ( pull1n ));
thand0  C11_U (.y ( f_N3 ) , .d ( f_state[0] ) , .c ( f_state[1] ) , .b ( t_state[0] ) , .a ( t_state[1] ));
th22  C11_U_0 (.y ( t_N3 ) , .b ( f_state[0] ) , .a ( f_state[1] ));
thand0  C13_U (.y ( t_N4 ) , .d ( t_state[0] ) , .c ( f_state[1] ) , .b ( f_state[0] ) , .a ( t_state[1] ));
th22  C13_U_0 (.y ( f_N4 ) , .b ( t_state[0] ) , .a ( f_state[1] ));
thand0  C16_U (.y ( t_N6 ) , .d ( f_state[0] ) , .c ( t_state[1] ) , .b ( t_state[0] ) , .a ( f_state[1] ));
th22  C16_U_0 (.y ( f_N6 ) , .b ( f_state[0] ) , .a ( t_state[1] ));
thand0  C34_U (.y ( t_N8 ) , .d ( t_N6 ) , .c ( f_N3 ) , .b ( f_N6 ) , .a ( t_N3 ));
th22  C34_U_0 (.y ( f_N8 ) , .b ( t_N6 ) , .a ( f_N3 ));
thand0  C38_U (.y ( t_N9 ) , .d ( t_N6 ) , .c ( t_N4 ) , .b ( f_N6 ) , .a ( f_N4 ));
th22  C38_U_0 (.y ( f_N9 ) , .b ( t_N6 ) , .a ( t_N4 ));
drlatn  \state_reg_1__0_U (.rsb ( reset ) , .ackout ( n2_N ) , .ackin ( acknet2 ) , .f_d ( f_q2_N ) , .t_d (
drlatr  \state_reg_1__0_U_0 (.rsb ( reset ) , .ackout ( n1_N ) , .ackin ( n2_N ) , .f_d ( f_q1_N ) , .t_d ( t
drlatn  \state_reg_1__0_U_1 (.rsb ( reset ) , .ackout ( acknet0 ) , .ackin ( n1_N ) , .f_d ( f_N9 ) , .t_d (
drlatn  \state_reg_0__0_U (.rsb ( reset ) , .ackout ( n2_N_0 ) , .ackin ( acknet2 ) , .f_d ( f_q2_N_0 ) , .t_
drlatr  \state_reg_0__0_U_0 (.rsb ( reset ) , .ackout ( n1_N_0 ) , .ackin ( n2_N_0 ) , .f_d ( f_q1_N_0 ) , .t
drlatn  \state_reg_0__0_U_1 (.rsb ( reset ) , .ackout ( acknet1 ) , .ackin ( n1_N_0 ) , .f_d ( f_N8 ) , .t_d
```

Figure 34: Binary State-Machine State Sequencer UNCLE Result
(part of Verilog net-list)

```
//    Nell_Version_14_05_20_08_16__
TH33N_1x state_machine_0_0_ENBRANK( .Z ( s_0_[0] ), .A ( ns_0_[0] ), .B ( rega_ack_0_ ), .C ( ack_in_0_ ), .I(reset___) );
TH33N_1x state_machine_0_1_ENBRANK( .Z ( s_0_[1] ), .A ( ns_0_[1] ), .B ( rega_ack_0_ ), .C ( ack_in_0_ ), .I(reset___) );
TH33N_1x state_machine_0_2_ENBRANK( .Z ( s_0_[2] ), .A ( ns_0_[2] ), .B ( rega_ack_0_ ), .C ( ack_in_0_ ), .I(reset___) );
TH33N_1x state_machine_0_3_ENBRANK( .Z ( s_0_[3] ), .A ( ns_0_[3] ), .B ( rega_ack_0_ ), .C ( ack_in_0_ ), .I(reset___) );
TH22N_1x state_machine_0_4_ENBRANK( .Z ( ns_0_[0] ), .A ( regb_0_[0] ), .B ( s_ack_0_ ), .I(reset___) );
TH22N_1x state_machine_0_5_ENBRANK( .Z ( ns_0_[1] ), .A ( regb_0_[1] ), .B ( s_ack_0_ ), .I(reset___) );
TH22N_1x state_machine_0_6_ENBRANK( .Z ( ns_0_[2] ), .A ( regb_0_[2] ), .B ( s_ack_0_ ), .I(reset___) );
TH22N_1x state_machine_0_7_ENBRANK( .Z ( ns_0_[3] ), .A ( regb_0_[3] ), .B ( s_ack_0_ ), .I(reset___) );
TH22N_1x state_machine_0_8_ENBRANK( .Z ( rega_0_[0] ), .A ( s_0_[3] ), .B ( regb_ack_0_ ), .I(reset___) );
TH22N_1x state_machine_0_9_ENBRANK( .Z ( rega_0_[1] ), .A ( s_0_[0] ), .B ( regb_ack_0_ ), .I(reset___) );
TH22N_1x state_machine_0_10_ENBRANK( .Z ( rega_0_[2] ), .A ( s_0_[1] ), .B ( regb_ack_0_ ), .I(reset___) );
TH22N_1x state_machine_0_11_ENBRANK( .Z ( rega_0_[3] ), .A ( s_0_[2] ), .B ( regb_ack_0_ ), .I(reset___) );
TH22D_1x state_machine_0_12_ENBRANK( .Z ( regb_0_[0] ), .A ( rega_0_[0] ), .B ( ns_ack_0_ ), .I(reset___) );
TH22N_1x state_machine_0_13_ENBRANK( .Z ( regb_0_[1] ), .A ( rega_0_[1] ), .B ( ns_ack_0_ ), .I(reset___) );
TH22N_1x state_machine_0_14_ENBRANK( .Z ( regb_0_[2] ), .A ( rega_0_[2] ), .B ( ns_ack_0_ ), .I(reset___) );
TH22N_1x state_machine_0_15_ENBRANK( .Z ( regb_0_[3] ), .A ( rega_0_[3] ), .B ( ns_ack_0_ ), .I(reset___) );
T11B_1x state_machine_0_16_COMPINV( .Z ( rega_ack_0_ ), .A ( \\0_0_  ) );
T11B_1x state_machine_0_17_COMPINV( .Z ( regb_ack_0_ ), .A ( \\1_0_  ) );
T11B_1x state_machine_0_18_COMPINV( .Z ( ns_ack_0_ ), .A ( \\2_0_  ) );
T11B_1x state_machine_0_19_COMPINV( .Z ( s_ack_0_ ), .A ( \\3_0_  ) );
T14_1x state_machine_0_20( .Z ( \\0_0_  ), .A ( rega_0_[0] ), .B ( rega_0_[1] ), .C ( rega_0_[2] ), .D ( rega_0_[3] ) );
T14_1x state_machine_0_21( .Z ( \\1_0_  ), .A ( regb_0_[0] ), .B ( regb_0_[1] ), .C ( regb_0_[2] ), .D ( regb_0_[3] ) );
T14_1x state_machine_0_22( .Z ( \\2_0_  ), .A ( ns_0_[0] ), .B ( ns_0_[1] ), .C ( ns_0_[2] ), .D ( ns_0_[3] ) );
T14_1x state_machine_0_23( .Z ( \\3_0_  ), .A ( s_0_[0] ), .B ( s_0_[1] ), .C ( s_0_[2] ), .D ( s_0_[3] ) );
```

Figure 35: State Sequencer NELL result NCL net-list

3.2.3 "Monkey Get Banana" Logic with Simple State Sequencer

The *Monkey Get Banana* machine is a state-machine that controls a banana vending machine for a hypothetical monkey feeder [18]. The monkey has to push four buttons – A, B, C, D – in sequence to get a banana. If she pushes any button out of sequence, the machine is reset and she has to start with A again. Figure 36 is the circuit diagram designed using NCL threshold gates and the circuit is designed using Structural-Verilog on Figure 37.

As shown in Figure 38, in the functional simulation result of the machine, after pushing each button sequentially, the "cntl_out" result will generate a high signal and the monkey will then get the banana.

Finally, the two machines (Simple State Sequencer and *Monkey Get Banana*) were merged to one module and their control signals connected (Figure 39). Figure 40 represents the simulation result of the merged module, which is functionally correct and operates without a centralized clock. Figure 41 reports the Altera/Intel SignalTap signal monitoring result on a Terasic

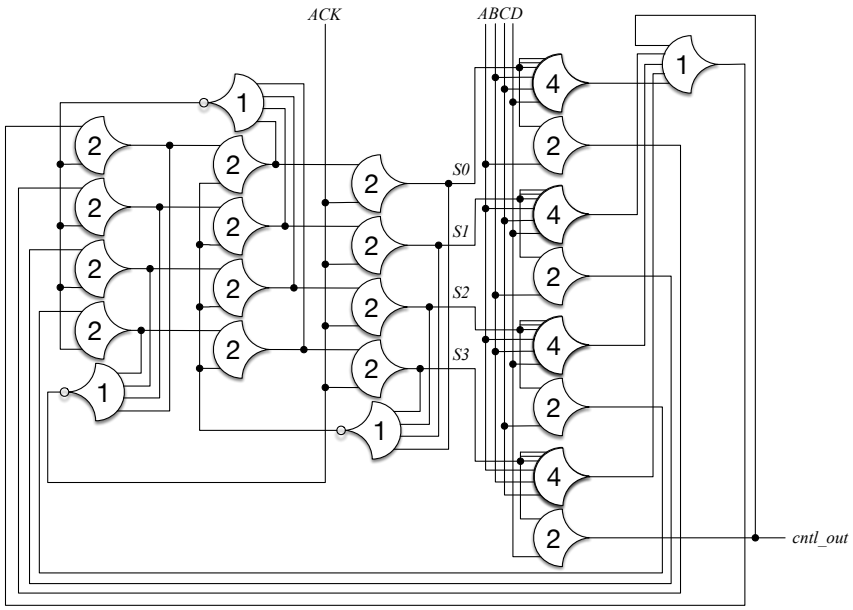

Figure 36: Monkey Get Banana State Machine - redrawn from [18]

```
// Module call
reg_1bit_init1 reg_1bit_1(reset, th15_s0 ,th14b_x2 ,th14b_x3 ,th14b_x1 ,th22_11 ,th22_12 ,th22_13);
reg_1bit_init0 reg_1bit_2(reset, th22_s1 ,th14b_x2 ,th14b_x3 ,th14b_x1 ,th22_21 ,th22_22 ,th22_23);
reg_1bit_init0 reg_1bit_3(reset, th22_s2 ,th14b_x2 ,th14b_x3 ,th14b_x1 ,th22_31 ,th22_32 ,th22_33);
reg_1bit_init0 reg_1bit_4(reset, th22_s3 ,th14b_x2 ,th14b_x3 ,th14b_x1 ,th22_41 ,th22_42 ,th22_43);

th14b th14b_1(th14b_x1 ,th22_11 ,th22_21 ,th22_31 ,th22_41);
th14b th14b_2(th14b_x2 ,th22_12 ,th22_22 ,th22_32 ,th22_42);
th14b th14b_3(th14b_x3 ,th22_13 ,th22_23 ,th22_33 ,th22_43);

th44w3 th44w3_1(th44w3_s0 ,th22_13 ,B ,C ,D);
th22r th22_1(th22_s1 ,th22_13 ,A, reset);
th44w3 th44w3_2(th44w3_s1 ,th22_23 ,A ,C ,D);
th22r th22_2(th22_s2 ,th22_23 ,B, reset);
th44w3 th44w3_3(th44w3_s2 ,th22_33 ,A ,B ,D);
th22r th22_3(th22_s3 ,th22_33 ,C, reset);
th44w3 th44w3_4(th44w3_s3 ,th22_43 ,A ,B ,C);
th22r th22_4(th22_s4 ,th22_43 ,D, reset);

th15 th15_1(th15_s0 ,th22_s4, th44w3_s0 ,th44w3_s1 ,th44w3_s2 ,th44w3_s3);
```

Figure 37: Structural-Verilog Design of Monkey Get Banana

DE-0 board, which is also working correctly, showing that all signal transitions are fully logically determined. The signal transition speed on this Cyclone-IV device is about 100MHz, and is fully asynchronous and logically working without the clock.

3.2.4 Implementation result on Cyclone-IV FPGA

Table 6 shows logic resource usage comparison results of each application. The applications were tested on the DE-0 board Altera/Intel Cyclone-IV FPGA. The NELL library supports only the User-Defined Primitive (UDP) model for its simulation therefore the NELL design type is only tested by using the UDP model on the Cyclone-IV. The Structural-Verilog and UNCLE versions were also tested using the UDP model to compare with the NELL design

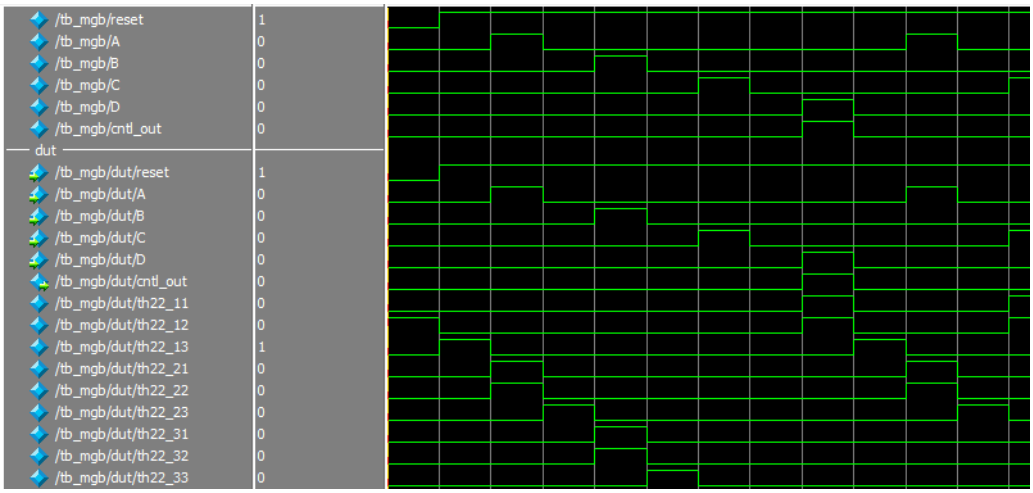

Figure 38: Functional Simulation of Monkey Get Banana

```
// Module call
sss sss( // Simple State Sequencer
        .reset(reset),
        .ack_in(ack),
        .s0(s0),
        .s1(s1),
        .s2(s2),
        .s3(s3)
        );

mgb mgb( // Monkey Get Banana
        .reset(reset),
        .A(s0),
        .B(s1),
        .C(s2),
        .D(s3),
        .cntl_out(give_banana),
        .ack_out(ack)
        );
```

Figure 39: Merge Simple State Sequencer and Monkey Get Banana

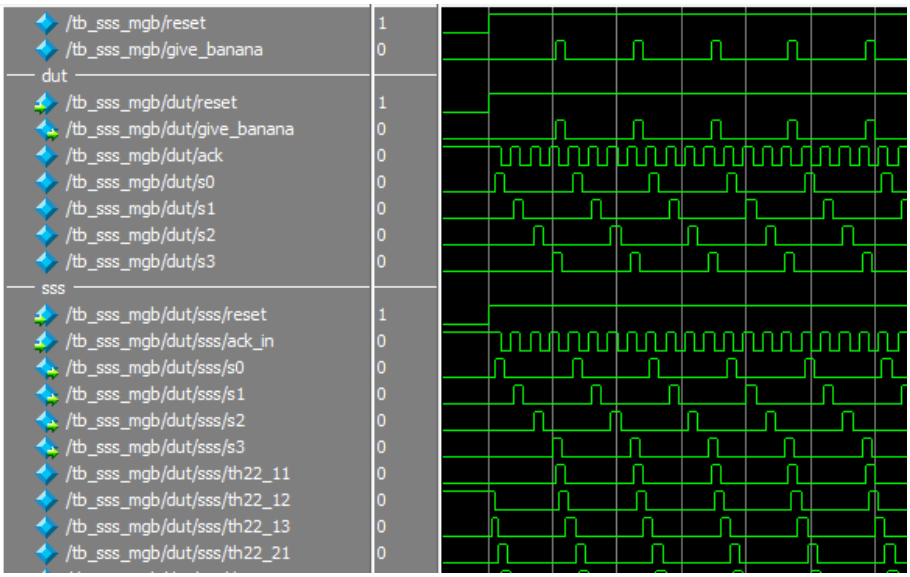

Figure 40: Functional Simulation of Simple State Sequencer and Monkey Get Banana

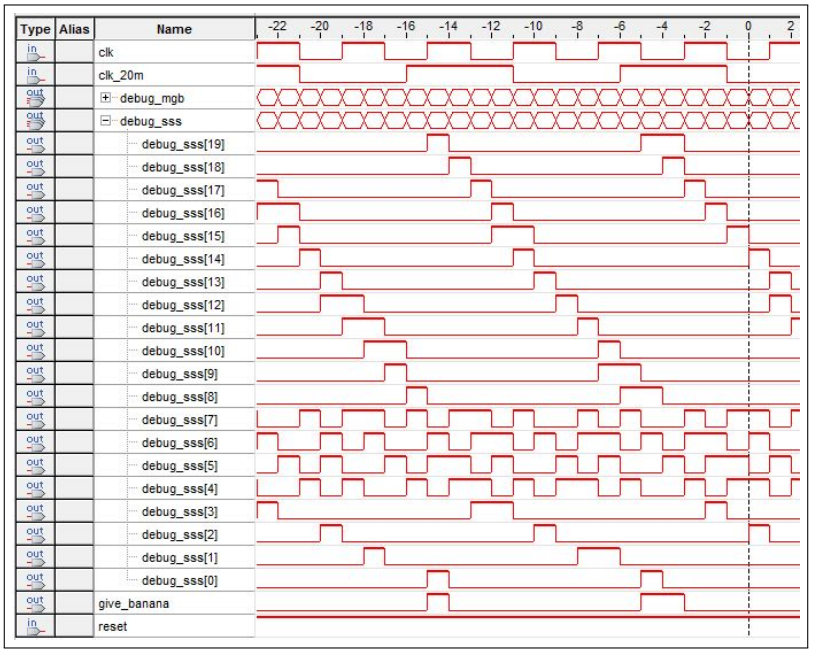

Figure 41: Simple State Sequencer and Monkey Get Banana- SignalTap monitoring

style. The design was also tested using a Boolean gate library. The Boolean gate library uses fewer Logic Elements compared to other library types (the FPGA implementation models are introduced in Section 3.4).

Table 6: Logic Resource Usage Comparison Report

Application	Design Style	Threshold Gates Counts	Logic Element(LE) Usage	
			User-Defined Primitive (UDP) Library	Boolean Gate Library
1-Bit Full Adder with Registers	Structural Verilog	21	50	21
	UNCLE	51	110	51
	NELL	39	77	-
	NELL Optimization	23	50	-
Simple State Sequencer	Structural Verilog	20	54	24
	UNCLE-One-Hot	84	213	104
	UNCLE-Binary	29	71	33
	NELL	24	53	-

3.3 ASIC Design Implementation for NCL Circuits

It is possible to reuse clocked Boolean gates for an NCL design as described by Fant in [18]. However, this approach is inefficient and will, at best, be useful only to generate a simulation model for testing or for a prototype FPGA implementation. Therefore, a specific NCL cell library will be necessary to support an ASIC flow. An ASIC (Application Specific Integrated Circuit) design flow can be separated into three different parts: Cell Library design,

Front-End design and Back-End design, which are explained in the following sections.

3.3.1 NCL Cell Library Design

In this section, the flow that was used for to derive NCL Cell Library design will be explained. The library used in this work was built using a 28nm fully depleted silicon on insulator CMOS process (28FDSOI) sourced from ST-Microelectronics via CMP [102]. As discussed in the Section 2.2.3.3, NCL is typically designed using a restricted sub-set of 27 fundamental majority logic (threshold) gates. The NCL Cell Library design process, in itself, is not much different from the normal clocked Boolean cell design that is typically supported by foundries such as TSMC, Samsung, Intel, Global Foundry and ST Microelectronics etc.

Table 7 shows the Boolean equations of the cells that were designed for this NCL implementation. Each equation is calculated from the threshold requirements of the gates in Table 2 and in Section 2.2.3.3 for Set/Reset/Hold1/Hold0. The equations for a *Static* NCL cell library (Table 7) effectively divide into two NMOS networks for Set and Hold1, and two in PMOS for the Reset and Hold0. The equations can be further combined and optimised to form a Set + Hold1 NMOS network and a Reset + Hold0 PMOS network. The optimization is quite flexible and can be changed based on the transistor placement and layout.

A number of alternative cell styles are possible, including Semi-Static, MTNCL [19], Differential [103], [104] and so on[3]. Because the static design is the most robust against noise, especially when the supply voltages are near or at sub-threshold, we used only the static NCL cell library in this work. It can be seen that the table actually has 37 cells. The ten cells included in addition to the fundamental cells are register cells with initialisation (set/reset) inputs and also buffer/inverter cells.

Figure 42 shows NCL Cell library Design Flow. This flow needs the same PDKs (Process Design Kits) from the foundry, transistor Spice simulation models and Design Rule Check (DRC) files for physical verification. It also requires behavioral descriptions to support cell characterization. The flow begins with Schematic Capture (Cadence Virtuoso Composer) and Spice simulation based on the equations in Table 7. The NCL Cell Schematic Capture designs are functionally verified using digital simulation tools after exporting the schematic to Verilog

[3]This NCL Cell Library was designed with assistance from Master of Engineering students Jeeson Kim and Amith Babu Pulakkavil. Jeeson Kim built the Static NCL library and Amith Pulakkavil the MTNCL library. Later, a Semi-Static library was added by an undergraduate student, Phi-Hung Nguyen

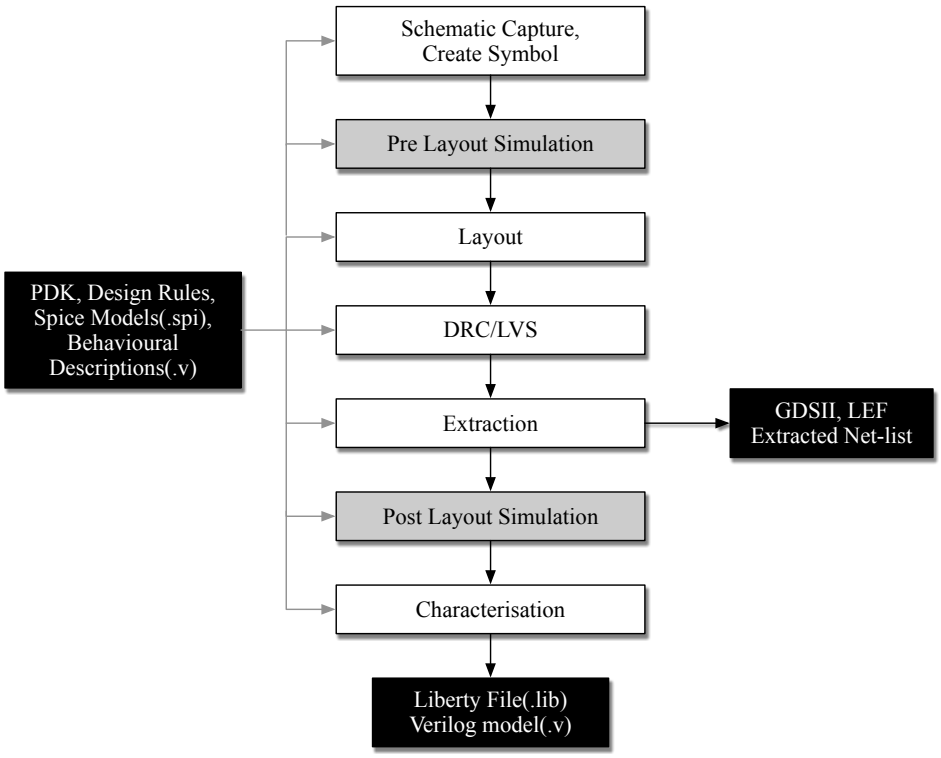

Figure 42: NCL Cell Library Design Flow

net-list. After successful Spice simulation, the schematic can be transferred to a layout (Cadence Layout-XL) using the same layout templates used by ST-Microelectronics for their Clocked Boolean library. DRC (Design Rule Check) and LVS (Layout vs Schematic) physical verification was performed using Mentor Graphics Calibre IC Verification and Signoff tools. The Layout was then extracted using Star-RC and the physical library files (GDSII and LEF) generated along with the extracted Spice net-list of the cells. Finally, Liberty files were derived using the characterization tools and Verilog simulation models created for back-annotation simulation. By using the same cell layout template, it was possible to use height-matched elements from the clocked standard cell library, such as Buffers, Inverters, AND and OR gates, within the NCL design.

Figure 43 shows the schematic view of a TH34W22 gate. The pmos (red color) and nmos (blue color) networks are matched with Table 7 (line number 18) TH34W22 cell equation with the optimized pmos equation: *AB(CD+Z)+ZCD(A+B)* and the optimized nmos equation: *AB+(A+B+Z)(c+D)+Z(A+B)*, when *Z* is the feed back to *y* in this schematic. Figure 44 shows the NCL Cell Layout of the extracted view of a TH34W22 cell[4]. This layout used 1.2μm cell

[4]This Cell Library design work, particularly the Leakage Power control and analysis has been published as "Static leakage control in null convention logic standard cells in 28 nm UTBB-FDSOI CMOS" [105]

	Gates	Boolean Equivalent	Pin Name	Set nmos Network	Hold1 (Complement of Reset)*Z nmos Network	Reset pmos Network	Hold0 (Complement of Set)*Z pmos Network	nmos Equation (Set+Hold1)	Optimized nmos Equation (Set+Hold1)	pmos Equation (Reset+Hold0)	Optimized pmos Equation (Reset+Hold0)	Optimized Transistor Count (pmos +nmos +inv)
1	TH12	OR2	Z,A,B	A+B		AB		A+B		AB		6
2	TH22		Z,A,B	AB	(A+B)Z	AB	(A+B)Z	AB+(A+B)Z	AB+(A+B)Z	AB+(A+B)Z	AB+(A+B)Z	12
3	TH13	OR3	Z,A,B,C	A+B+C		ABC		A+B+C		ABC		8
4	TH23		Z,A,B,C	AB+AC+BC	(A+B+C)Z	ABC	((A+B)(A+C)(B+C))Z	(AB+AC+BC)+(A+B+C)Z	AB+(A+B)(C+Z)+CZ	ABC+((A+B)(A+C)(B+C))Z	AB(C+Z)+(A+B)CZ	18
5	TH33		Z,A,B,C	ABC	(A+B+C)Z	ABC	(A+B+C)Z	ABC+(A+B+C)Z	ABC+Z(A+B+C)	ABC+(A+B+C)Z	ABC+Z(A+B+C)	16
6	TH23W2		Z,A,B,C	A+BC	(A+B+C)Z	ABC	(A(B+C))Z	(A+BC)+(A+B+C)Z	BC+A+Z(B+C)	ABC+(A(B+C))Z	A(B(C+Z)+CZ)	14
7	TH33W2		Z,A,B,C	AB+AC	(A+B+C)Z	ABC	((A+B)(A+C))Z	(AB+AC)+(A+B+C)Z	(B+C)(A+Z)+AZ	ABC+((A+B)(A+C))Z	BC(A+Z)+AZ	14
8	TH14	OR4	Z,A,B,C,D	A+B+C+D		ABCD		A+B+C+D		ABCD		10
9	TH24		Z,A,B,C,D	AB+AC+AD+BC+BD+CD	(A+B+C+D)Z	ABCD	((A+B)(A+C)(A+D)(B+C)(B+D)(C+D))Z	(AB+AC+AD+BC+BD+CD)+(A+B+C+D)Z	AB+(B+D)(C+D+Z)+C(D+Z)+DZ	ABCD+((A+B)(A+C)(A+D)(B+C)(B+D)(C+D))Z	AB(C(D+Z)+DZ)+ZCD(A+B)	26
10	TH34		Z,A,B,C,D	ABC+ABD+ACD+BCD	(A+B+C+D)Z	ABCD	((A+B+C)(A+B+D)(A+C+D)(B+C+D))Z	(ABC+ABD+ACD+BCD)+(A+B+C+D)Z	(AB+Z)(C+D)+(A+B)(CD+Z)	ABCD+((A+B+C)(A+B+D)(A+C+D)(B+C+D))Z	AB(CD+Z)+Z((A+B)(C+D)+CD))	24
11	TH44		Z,A,B,C,D	ABCD	(A+B+C+D)Z	ABCD	(A+B+C+D)Z	ABCD+(A+B+C+D)Z	ABCD+Z(A+B+C+D)	ABCD+(A+B+C+D)Z	ABCD+Z(A+B+C+D)	20
12	TH24W2		Z,A,B,C,D	A+BC+BD+CD	(A+B+C+D)Z	ABCD	(A(B+C)(B+D)(C+D))Z	(A+BC+BD+CD)+(A+B+C+D)Z	A+(B+Z)(C+D)+CD+ZB	ABCD+(A(B+C)(B+D)(C+D))Z	A(BC(D+Z)+ZD(B+C))	20
13	TH34W2		Z,A,B,C,D	AB+AC+AD+BCD	(A+B+C+D)Z	ABCD	((A+B)(A+C)(A+D)(B+C+D))Z	(AB+AC+AD+BCD)+(A+B+C+D)Z	(A+Z)(B+C+D)+BCD+ZA	ABCD+((A+B)(A+C)(A+D)(B+C+D))Z	BCD(A+Z)+AZ(B+C+D)	22
14	TH44W2		Z,A,B,C,D	ABC+ABD+ACD	(A+B+C+D)Z	ABCD	((A+B+C)(A+B+D)(A+C+D))Z	(ABC+ABD+ACD)+(A+B+C+D)Z	A(B(C+D)+CD+Z)+Z(B+C+D)	ABCD+((A+B+C)(A+B+D)(A+C+D))Z	CD(AB+Z)+Z(A+B(C+D))	23
15	TH34W3		Z,A,B,C,D	A+BCD	(A+B+C+D)Z	ABCD	(A(B+C+D))Z	(A+BCD)+(A+B+C+D)Z	A+BCD+B+C+DZ	ABCD+(A(B+C+D))Z	A(BCD+Z(B+C+D))	18
16	TH44W3		Z,A,B,C,D	AB+AC+AD	(A+B+C+D)Z	ABCD	((A+B)(A+C)(A+D))Z	(AB+AC+AD)+(A+B+C+D)Z	(A+Z)(B+C+D)+ZA	ABCD+((A+B)(A+C)(A+D))Z	BCD(A+Z)+AZ	16
17	TH24W22		Z,A,B,C,D	A+B+CD	(A+B+C+D)Z	ABCD	(AB(C+D))Z	(A+B+CD)+(A+B+C+D)Z	A+B+CD+Z(C+D)	ABCD+(AB(C+D))Z	AB(CD+Z(C+D))	16
18	TH34W22		Z,A,B,C,D	AB+AC+AD+BC+BD	(A+B+C+D)Z	ABCD	((A+B)(A+C)(A+D)(B+C)(B+D))Z	(AB+AC+AD+BC+BD)+(A+B+C+D)Z	AB+(A+B+Z)(C+D)+Z(A+B)	ABCD+((A+B)(A+C)(A+D)(B+C)(B+D))Z	AB(CD+Z)+ZCD(A+B)	22
19	TH44W22		Z,A,B,C,D	AB+ACD+BCD	(A+B+C+D)Z	ABCD	(A+B)(A+C+D)(B+C+D))Z	(AB+ACD+BCD)+(A+B+C+D)Z	(A+B)(CD+Z)+AB+Z(C+D)	ABCD+(A+B)(A+C+D)(B+C+D))Z	AB(CD+Z)+Z(A+B)(C+D)	22
20	TH54W22		Z,A,B,C,D	ABC+ABD	(A+B+C+D)Z	ABCD	((A+B+C)(A+B+D))Z	(ABC+ABD)+(A+B+C+D)Z	(AB+Z)(C+D)+Z(A+B)	ABCD+((A+B+C)(A+B+D))Z	CD(AB+Z)+Z(A+B)	18
21	TH34W32		Z,A,B,C,D	A+BC+BD	(A+B+C+D)Z	ABCD	(A(B+C)(B+D))Z	(A+BC+BD)+(A+B+C+D)Z	A+(C+D)(B+Z)+BZ	ABCD+(A(B+C)(B+D))Z	ACD(B+Z)+ABZ	17
22	TH54W32		Z,A,B,C,D	AB+ACD	(A+B+C+D)Z	ABCD	((A+B)(A+C+D))Z	(AB+ACD)+(A+B+C+D)Z	A(B+CD+Z)+BZ+C+D	ABCD+((A+B)(A+C+D))Z	A(BCD+Z)+BZ(C+D)	20
23	TH44W322		Z,A,B,C,D	AB+AC+AD+BC	(A+B+C+D)Z	ABCD	((A+B)(A+C)(A+D)(B+C))Z	(AB+AC+AD+BC)+(A+B+C+D)Z	(A+Z)(B+C+D)+BC+ZA	ABCD+((A+B)(A+C)(A+D)(B+C))Z	BCD(A+Z)+ZA(B+C)	20
24	TH54W322		Z,A,B,C,D	AB+AC+BCD	(A+B+C+D)Z	ABCD	((A+B)(A+C)(B+C+D))Z	(AB+AC+BCD)+(A+B+C+D)Z	(A+D)(Z+BC)+(B+C)(A+Z)	ABCD+((A+B)(A+C)(B+C+D))Z	BC(AD+Z)+ZA(B+C+D)	21
25	THXOR		Z,A,B,C,D	AB+CD	(A+B+C+D)Z	ABCD	((A+B)(C+D))Z	(AB+CD)+(A+B+C+D)Z	AB+CD+Z(A+B+C+D)	ABCD+((A+B)(C+D))Z	ABCD+Z((A+B)(C+D))	20
26	THAND		Z,A,B,C,D	AB+BC+AD	(A+B+C+D)Z	ABCD	((A+B)(B+C)(A+D))Z	(AB+BC+AD)+(A+B+C+D)Z	(B+D)(A+Z)+BC+Z(A+C)	ABCD+((A+B)(B+C)(A+D))Z	BD(Z+AC)+AZ(B+C)	20
27	THCOMP		Z,A,B,C,D	AC+BC+AD+BD	(A+B+C+D)Z	ABCD	((A+C)(B+C)(A+D)(B+D))Z	(AC+BC+AD+BD)+(A+B+C+D)Z	(A+B)(D+C+Z)+Z(C+D)	ABCD+((A+C)(B+C)(A+D)(B+D))Z	AB(CD+Z)+CDZ	18
28	TH22D		Z,A,B,I	AB+I	(A+B+I)Z	ABI	(A+B)IZ	AB+I+(A+B+I)Z	AB+I+(A+B+I)Z	ABI+(A+B)IZ	I(AB+(A+B)Z)	14
29	TH22N		Z,,A,B,I	AB+I′	(A+B)Z	ABI′	(A+B)I′Z	AB+I′+(A+B)Z	AB+I′+(A+B)Z	ABI′+(A+B)I′Z	I′(AB+(A+B)Z)	16
30	TH33D		Z,A,B,C,I	ABC+I	(A+B+C)Z	ABCI	(A+B+C)IZ	ABC+I+(A+B+C)Z	ABC+I+(A+B+C)Z	ABCI+(A+B+C)IZ	I(ABC+(A+B+C)Z)	16
31	TH33N		Z,A,B,C,I	ABC+I′	(A+B+C)Z	ABCI′	(A+B+C)I′Z	ABC+I′+(A+B+C)Z	ABC+I′+(A+B+C)Z	ABCI′+(A+B+C)I′Z	I′(ABC+(A+B+C)Z)	18
32	TH11B	INV	ZN,A					ZN=!A				2
33	TH11BN	NOR2	ZN,A,I					ZN=!(A+I)				4
34	TH11BD	IND2	ZN,A,I					ZN=!(A*I)				6
35	TH11N	INR2	Z,A,I					Z=!(!A+I)				8
36	TH11D	OR2	Z,A,I					Z=A+I				6
37	TH12B	NOR2	ZN,A,B					ZN=!(A+B)				4

Table 7: NCL Static Cell Library Equations

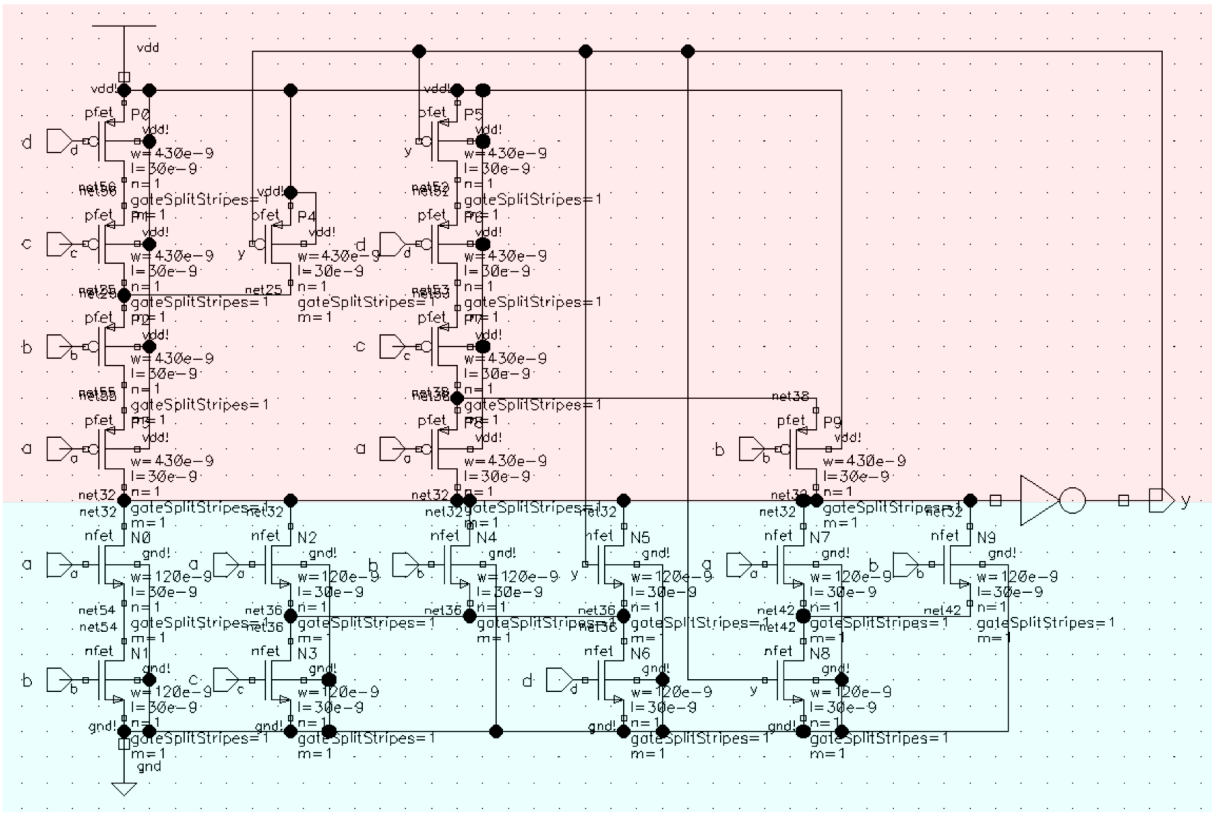

Figure 43: Cell Schematic View of TH34W22

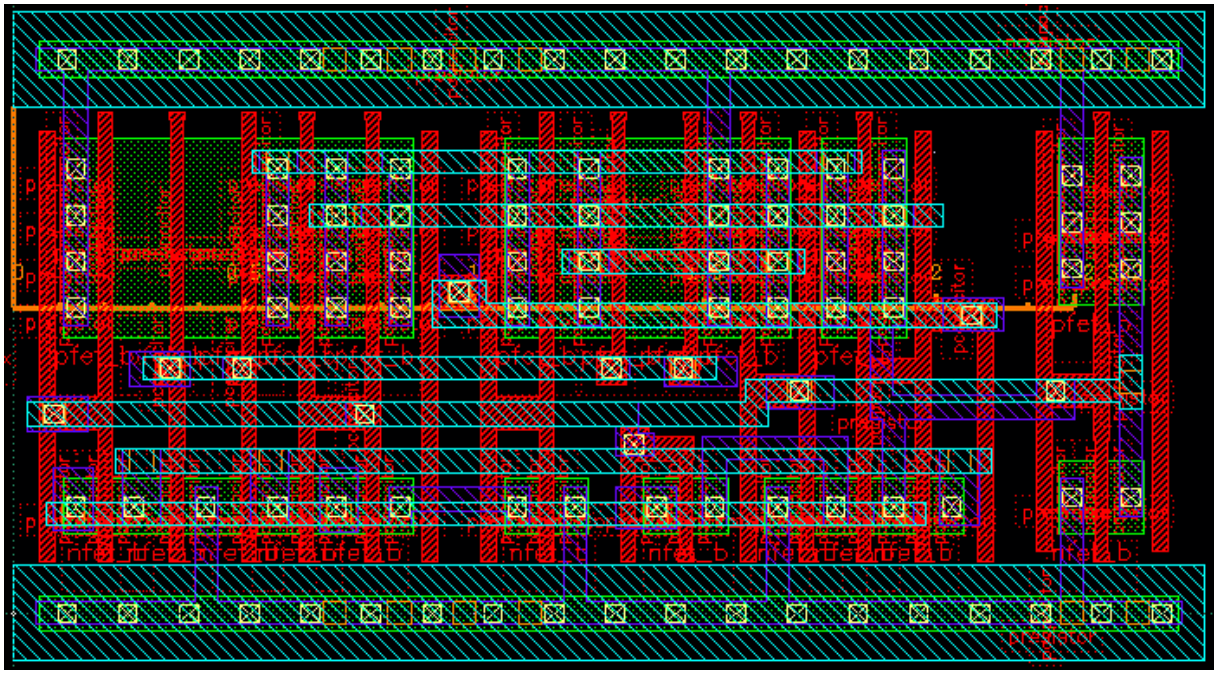

Figure 44: Cell Layout Extracted View of TH34W22

height and 2.584 μm width (N x 0.136) and also 0.208 μm for the V_{DD} and V_{SS} lines based on the ST-Microelectronics cell templates [102].

All the generated NCL cells have been functionally verified using Spice (Cadence Spectre) simulation and also digital simulation tools (NC-Verilog, in this case) after converting the schematic to a Verilog transistor-level net-list. The input vectors for Spice simulation and digital simulation tools were generated based on the classification of the NCL cell's monotonic transition table. The details of the functional verification for the NCL cells including conversion flows are discussed in Appendix-C and the test-bench files and vectors are shared on the github [106].

3.3.2 Front-end Design

As discussed above, UNCLE and NELL are design tools for NCL that can be used to generate gate-level net-lists. Many previous NCL design groups have used structural Verilog or VHDL while some groups have used special type-definitions or packages within System Verilog or VHDL. Obviously, it is also possible to use Schematic Capture tools and export to a Verilog NCL net-list from the schematic design. For small designs, any or all of those design methodologies can be used and by using Spice simulations the designs verified. However, as the design increases to the size and complexity of a CPU core design, more capable (semi)automatic design tools like UNCLE or NELL will be necessary.

As outlined previously in the Table 6, a brief comparison has been undertaken between UNCLE, NELL and Structural-Verilog approaches based on NCL theory using examples of both Data-Flow model (Full Adder) and Control-Flow model (Simple State-Machine). The example designs were implemented on an Altera/Intel Cyclone-IV device and the table also shows the logic resource usage comparison results for those examples. Physical testing (on FPGA hardware) was performed on a Terasic DE-0 board containing a Cyclone-IV FPGA. As the NELL library supports only the User-Defined Primitive (UDP) model for simulation, this was the only option tested using on NELL on the Cyclone-IV. The Structural-Verilog and UNCLE net-lists were also tested using the UDP model to compare with the NELL design style. Finally, the design was also tested using a Boolean Gate library derived from the 28nm FDSOI process kit.

The results indicate that, while the structural Verilog approach always results in the most compact design, NELL offers the most efficient design method for complex NCL applications. Further, the resulting code is simpler than a structural-Verilog design. On the other hand, the use of NELL requires more effort from the designer to learn NELL programming skills and to be able to understand NCL design theory more thoroughly.

3.3.3 Back-end Design

The existing synchronous back-end design flow was used because no dedicated back-end tools currently exist for NCL. The block-level design employed Cadence Virtuoso Schematic/Layout tools for the design and cell Layout and Spice/Spectre simulation tools for testing. However, for the CPU core described later, Auto Place and Route tools were used, along with sign-off timing and power analysis tools operating on the UNCLE and NELL net-lists targeting the NCL cell library. Standard logic simulation tools were used for logical verification and timing simulation. Figure 45 shows the design flow.

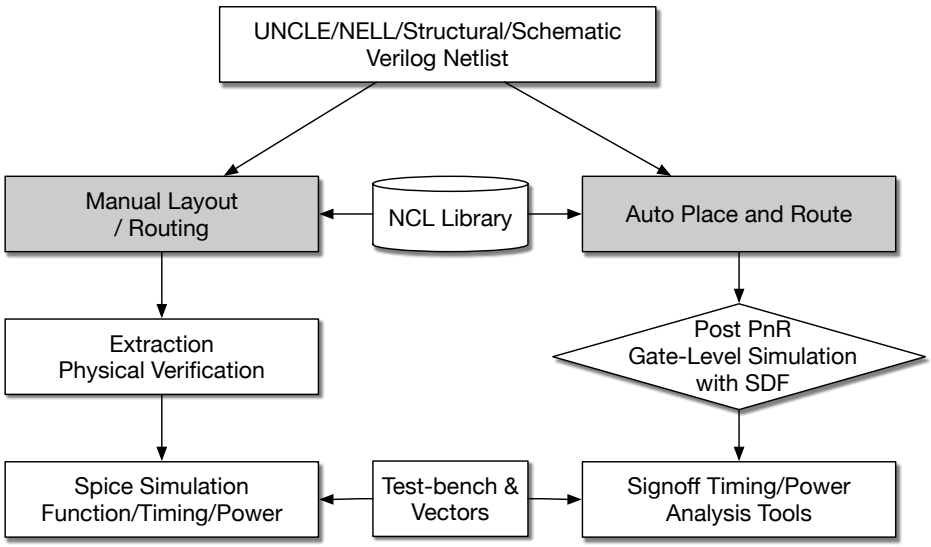

Figure 45: Back-end Design Flow

NCL Circuit AMS (Analog Mixed Signal) simulation with Verilog test-bench

As mentioned previously, NELL is used as the main Front-end design tool. The NELL compiled net-list files are imported to Cadence Virtuoso as schematic view files using the 28nm FDSOI NCL cell library. For an NCL circuit level simulation with handshake signals, it is necessary to carefully consider the Spice simulation because the input stimulus data of the Spice simulation is produced based on the output acknowledge signals so its acknowledge response timing is critical to the simulation. By this we mean that, to correctly model the input timing of an NCL

Figure 46: Ripple-Carry Adder Circuit-Level Simulation using Virtuoso-AMS simulator with 100 Input Vectors
(a) reset input (10nS), (b) input-vector A acknowledge, (c) input-vector A (64-Bit Dual-Rail), (d) input-vector B acknowledge, (e) input-vector B (64-Bit Dual-Rail), (f) output result acknowledge, (g) output result (64-Bit Dual-Rail)

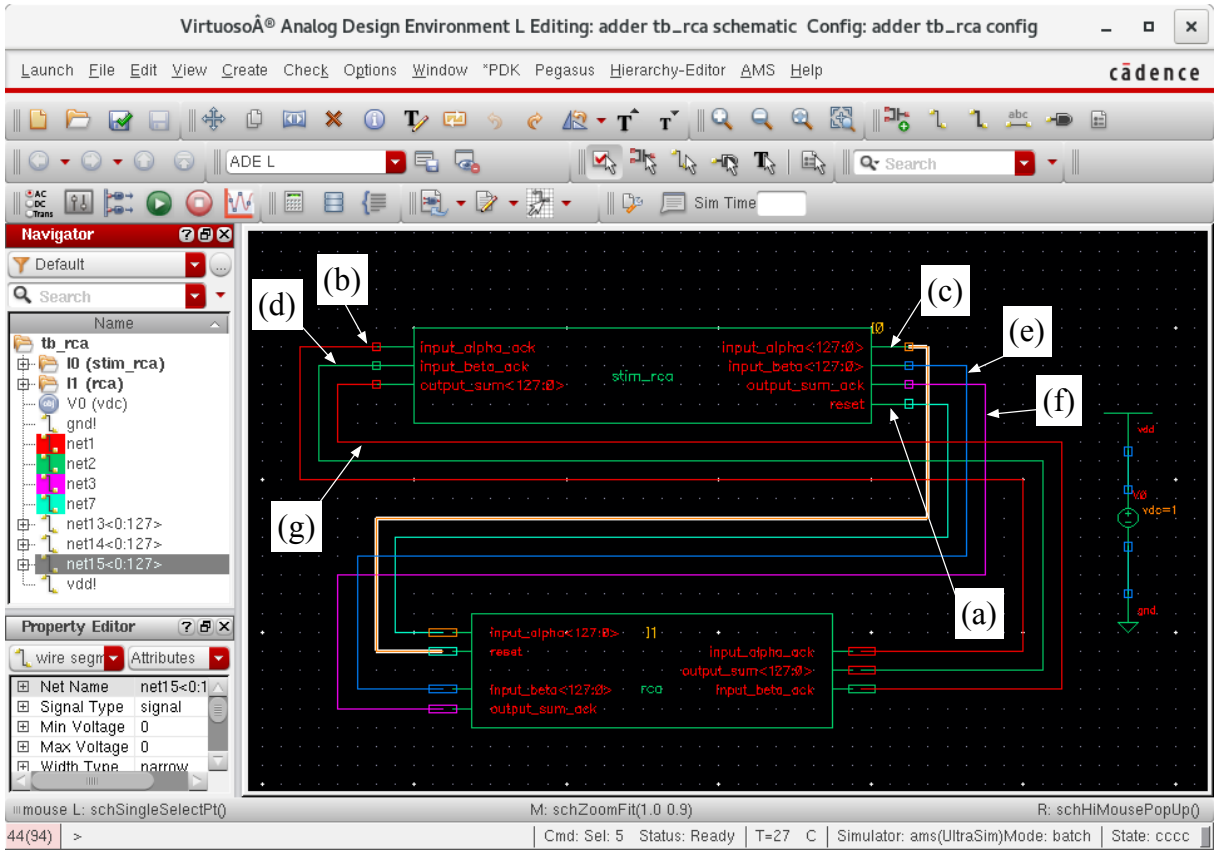

Figure 47: Ripple-Carry Adder Virtuoso-AMS Simulation Test-bench

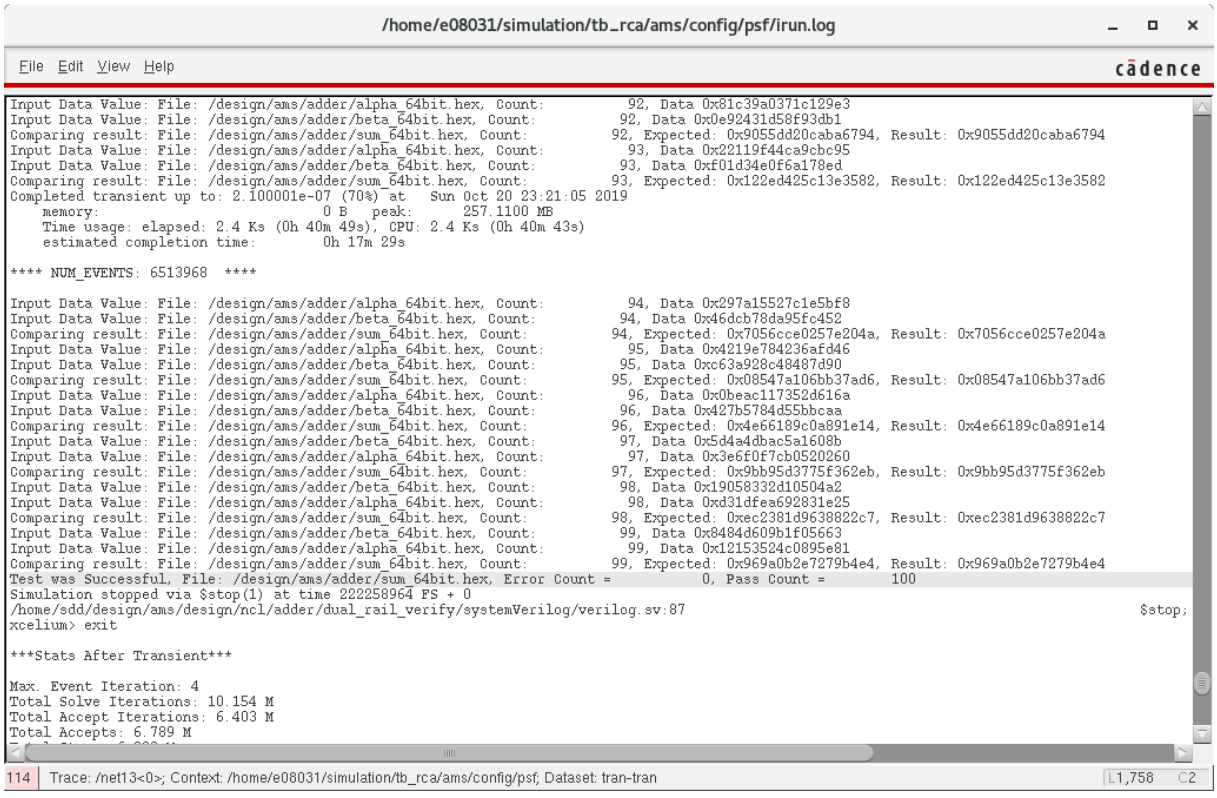

Figure 48: Ripple-Carry Adder Virtuoso-AMS Simulation Result Comparison

circuit, the testbench would have to respond to the handshaking signals received back from the circuit under test, something that is difficult to do in Spice. However, while general Spice simulation does not fully support this sort of handshaking protocol, there are several ways to use circuit level simulation to achieve the necessary stimulus/response outcome. One way is to use a Verilog (or VHDL)-A with Spice simulation tools (Spectre of Cadence or Hspice of Synopsys etc). The other is by using AMS (Analog Mixed Signal) tools with a Verilog test-bench and built-in connection libraries (i.e., abstract A/D, D/A converters). Figure 46 shows the example of 64-Bit Ripple-Carry Adder AMS simulation (Cadence Virtuoso-AMS simuation with Ultrasim full-chip simulator and NC-Verilog digital simulator) with 100 64-Bit pseudo-random input vectors. The performance results are averaged over the simulation time and vectors. Figure 47 shows the test-bench file for AMS simulation - the Stimuli Generator (stm_rca) and Ripple-Carry Adder circuit (rca). The wire groups (a) to (g) are matched with the waveform groups (a) to (g) of Figure 46. Figure 48 shows the 100 input vector simulation after execution/verification on the Verilog test-bench.

3.4 FPGA Implementation for an NCL Circuit

This section introduces an NCL implementation methodology aimed at commercial FPGA tools and devices[5]. These simulation experiments have been carried out targeting various Xilinx, Altera/Intel and Actel devices and tested on their physical FPGA boards.

3.4.1 FPGA Design Tool Flow for NCL Circuits

To implement NCL design on FPGA devices, there are five alternative types of gate descriptions: a Verilog behavioral model, Verilog LUT model, Verilog UDP model, Schematic design model and Verilog Boolean model. These have individual advantages and disadvantages, as will be discussed below.

rsb	a	b	y
0	x	x	1
1	0	0	0
1	0	1	Hold
1	1	0	Hold
1	1	1	1

Table 8: Truth Table of TH22S Gate

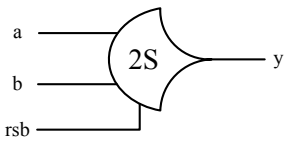

Figure 49: TH22S Gate Symbol

Here, the five different approaches to NCL description are illustrated using an example of a TH22S gate implementation on a Xilinx Virtex-6 FPGA. Figure 49 is the threshold gate

[5]This work has been published as "Design Techniques for NCL-Based Asynchronous Circuits on Commercial FPGA" [107]

symbol of TH22S and Table 8 shows the truth table of the gate. It performs the same function as a TH22 with an added 'rsb' signal for initialization.

• **Verilog Behavioral Model**: This technique uses the behavioral Verilog design of the UNCLE simulation model [15]. Similar behavioral models using VHDL have also been derived in [108]. It can be seen in Figure 50 that the design model for the static TH22S gate comprises four basic parts: reset, transition to DATA, transition to NULL and data hold. The reset part is included for initialization in the case where the gates are used to create an asynchronous latch. The final part handles the State Hold that maintains the current state when the input is in its hysteresis condition. Being a behavioral description, the actual logic resulting from this description type depends entirely on the synthesis tools.

```
// th22s - Verilog behavioral
module th22s(y,a,b,rsb);
   output y;
   input  a;
   input  b;
   input  rsb;

   reg yi;
   always @(a or b or rsb) begin
      if(rsb == 0) begin// reset
        yi <=  1;
      end
      else if(((a) & (b)))//Data
     begin
        yi <=  1;
     end
      else if(((a==0)&(b==0)))//Null
     begin
        yi <=  0;
     end// else State Hold
   end
   assign #1 y = yi;
endmodule
```

Figure 50: Behavioral Model of TH22S NCL Gate

```
// th22s - LUT
module th22s(y,a,b,rsb)
  output y;
   input a;
   input b;
   input rsb;

   wire yi;
   assign #1 y = yi;

   // LUT4: 4-input Look-Up Table
   //       Virtex-6
   // Xilinx HDL Language Template
   LUT4 #(// Specify LUT Contents
     .INIT(16'hE8FF)
     ) LUT4_inst (
      .O(yi),// LUT general output
      .I0(a),// LUT input
      .I1(b),// LUT input
      .I2(yi),// Feedback
      .I3(rsb)// reset
     );
endmodule
```

Figure 51: LUT-based Model of TH22S NCL Gate

• **Verilog LUT model**: Figure 51 is the Verilog LUT model. Commercial FPGA tools generally support the creation of a dedicated LUT function using a common hardware description language. The example of Figure 52 uses a 4-input LUT to implement the TH22S gate. The LUT contents (0xE8FF in this case) arise from a direct evaluation of the truth table (Table 9) for this gate. The TH22S gate is mapped to the LUT4 component with an external feedback connection from the LUT output to input I2.

• **Verilog UDP model**: The Verilog User Defined Primitive (UDP) function can be used to generate any type of user defined primitives, including basic Boolean gate functions.

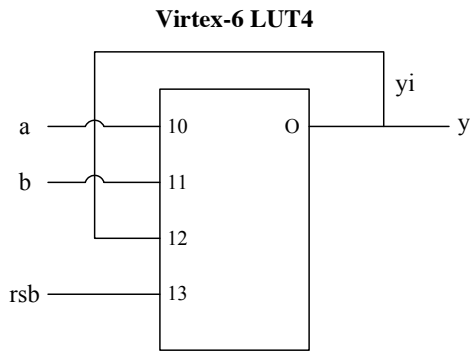

Figure 52: LUT Mapping of TH22S NCL Gate

I3	I2	I2	I0	O
rsb	yi	b	a	yi
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Table 9: Virtex-6 LUT Truth Table of TH22S NCL Gate

Traditionally, UDP has been used for verification purposes via test bench code but commercial FPGA vendors have now begun to support UDP as a synthesizable function. Verilog code for the UDP primitive describing the TH22S is shown in Figure 53. This is very useful in the case of NCL gate generation because the NCL gates can be set up as user defined primitives. Unlike the LUT case, UDP is part of the standard Verilog language therefore the design can target any of the FPGA vendors as long as their specific synthesis tool supports these UDP descriptions.

```
// th22s - UDP
module th22s(y,a,b,rsb);
   output y;
   input  a;
   input  b;
   input  rsb;

   wire   yi;
   assign #1 y = yi;
   //instantiate the udp
   TH22S_UDP I_TH22S_UDP(yi,a,b,rsb);
endmodule

primitive TH22S_UDP (Z, A, B, I);
   output Z;
   input  A, B, I; // I: "Active Low"
   reg     Z;
   initial Z = 0;
   table
      //A B I : current:next
      ? ? 0 : ? : 1;
      0 0 1 : ? : 0;
      0 1 1 : ? : -;
      1 0 1 : ? : -;
      1 1 1 : ? : 1;
   endtable
endprimitive
```

Figure 53: Verilog UDP Design Model of TH22S Gate

```
// th22s - Boolean gates
module th22s(y,a,b,rsb)
  output y;
   input a;
   input b;
   input rsb;

   wire  g1_out, g2_out, g3_out, g4_out
    , g5_out, g6_out, g7_out;

   // Gate Instantiation
   and G1 (g1_out, a, b);// Data
   nor G2 (g2_out, a, b);// Null

   not G3 (g3_out, rsb);// Active Low rsb
   or G4 (g4_out, g3_out, g1_out);
   and G5  (g5_out, rsb, g2_out);

   // SR Flip-Flop
   nor G6 (g6_out, g4_out, g7_out);
   nor G7 (g7_out, g5_out, g6_out);

   // Output Assignment
   assign #1 y = g7_out;

endmodule
```

Figure 54: Boolean based TH22S Gate

• **Verilog Boolean gate model**: The general Boolean gate description (Figure 54) is most useful in a structural design style where the gate instantiation can point to any FPGA vendor and/or device family without limitation and, equally, to an ASIC standard cell component.

• **Schematic design model**: The basic schematic design style for NCL was first proposed in [18], which describes a complete NCL gates design methodology using general Boolean gates including initialization. The schematic diagram of the TH22S (Figure 56) illustrates the concept. The module comprises three basic parts: the RS-Latch at its output to implement the hysteresis function; the initialization stage (Reset in this case) plus the input logic stage that defines the logic of the gate. Then the schematic designs are instantiated in the NCL Verilog gate modules (Figure 55).

```
// th22s - Schematic
module th22s(y,a,b,rsb);
   output y;
   input  a;
   input  b;
   input  rsb;

   wire   yi;
   assign #1 y = yi;

   TH22S_SCH INST_TH22S
     (
      .A        (a),
      .B        (b),
      .RSB      (rsb),
      .Z        (yi)
      );
endmodule
```

Figure 55: Xilinx schematic module instantiation of TH22S gate

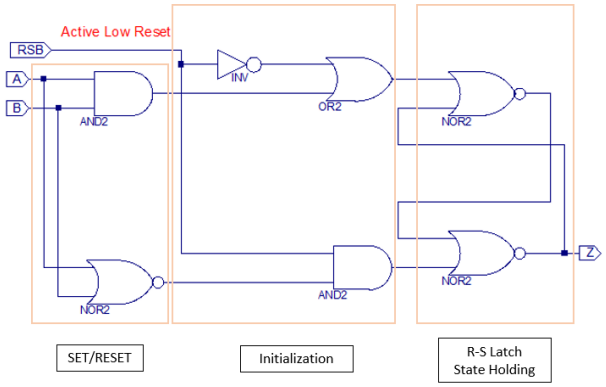

Figure 56: Xilinx schematic design of TH22S gate

Figure 57 shows the Test-Diagram on the FPGA including Altera/Intel SignalTap block for Signal Monitoring and Figure 58 shows the FPGA implementation of 8-Bit Up-Counter was created using the Boolean Gate model on the Altera/Intel Cyclone-IV device. The physical testing was done on a Terasic DE-0 board [6].

The five description methods shown above can be divided into two classes: vendor agnostic and vendor specific. It can be seen that the Behavioral, UDP and Boolean descriptions do not rely on a specific FPGA vendor or synthesis design tool because they are using standard Verilog library. In contrast, the LUT and schematic methods are specific to a particular chip type. The Behavioral Verilog and UDP models used separate Latch components internal to the CLB to implement the RS-Latch of each threshold gate but others used only LUTs.

[6]Other complex NCL design emulations on the FPGA board are presented in Appendix B "NCL FIR Filter Design on Commercial FPGA" and Chapter 5

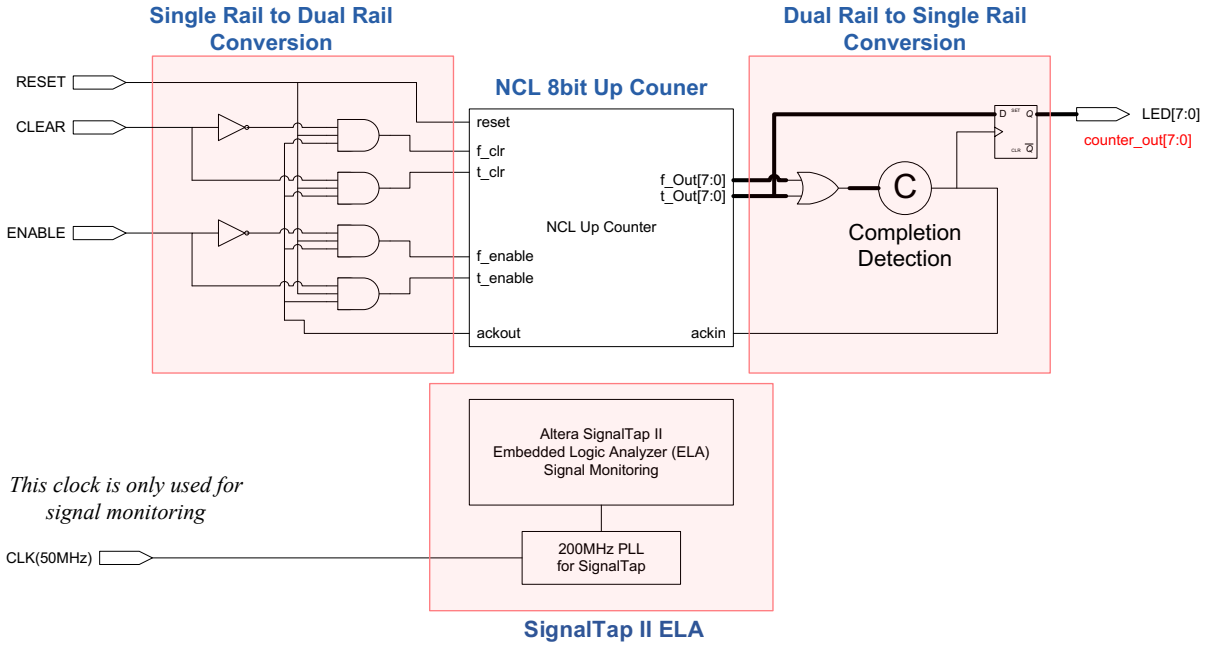

Figure 57: FPGA Up-Counter Design Test

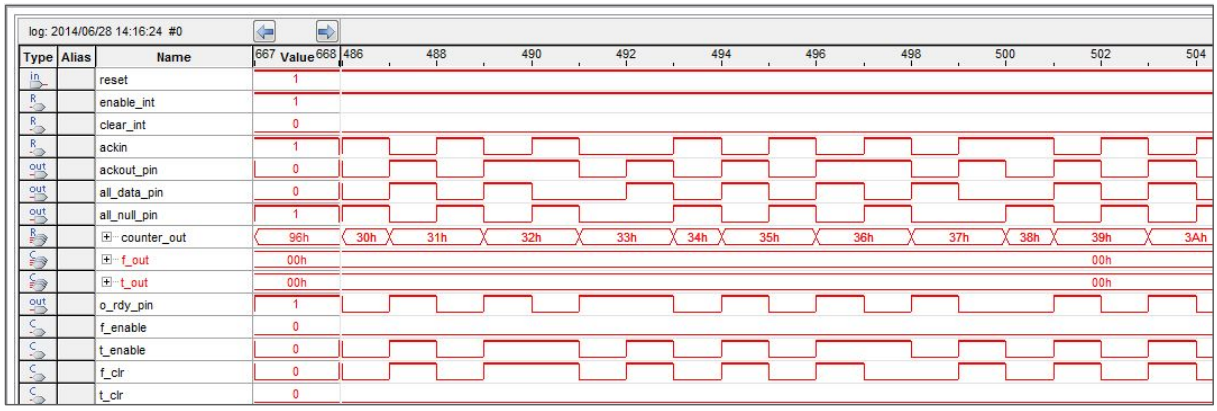

Figure 58: SignalTap waveform for NCL up-counter

3.5 Summary

In this chapter, the experimental tool flow that has been set up to support this research is described. As the great majority of current design tools and technologies support only clocked synchronous systems, it has been necessary to develop a specialized methodology aimed specifically at asynchronous Null Convention Logic design.

The tool flow has incorporated existing tools such as UNCLE from the University of Mississippi and the University of Arkansas as well as an industrial tool, NELL from Wave Computing. Some initial experiments have been performed based on both sequential control flow logic and a simple data flow logic to compare three alternative approaches to the hardware description of NCL circuits: Structural-Verilog, UNCLE and the NELL design. These

experiments compared the logic resources used in the net-lists resulting from each compiler. Structural-Verilog design is simply a text version of the schematic design and the logic can be manually designed and optimized using threshold gate instantiation. This style uses fewer logic resources but requires more effort because it is fully manual. In contrast, UNCLE simply converts an entirely synchronous Verilog design to an NCL net-list. It is therefore the easiest way to generate NCL functional modules and has the added advantage of supporting legacy ("dusty-deck") Verilog code modules. A major disadvantage of the approach is, because UNCLE converts the synchronous flip-flops to three-stage dual-rail registers, it generates significantly more logic gates in its net-list. Finally, NELL is a dedicated language for NCL. It conveniently expresses the NCL design and also generates a more efficient net-list, in that the code that is generated is somewhat simpler than that for the Structural-Verilog description. However, to program NELL effectively, an understanding of the NCL circuit architecture is required.

As a result of these experiments, NELL was selected as the most efficient of the available design method for complex NCL applications such as a CPU. On the other hand, to use NELL requires more effort to learn requisite programming skills and to understand NCL theory. On balance, and considering its efficiency advantages, the work in later chapters uses NELL extensively in the "front-end" stages (RTL design and simulation) of the CPU design process.

For the back-end of the flow, no special asynchronous design back-end tools currently exist, therefore it has been necessary to reuse a conventional synchronous back-end as well as its simulation and analysis tools. It has been shown that it is fairly straightforward to implement NCL designs on FPGA devices, although the resulting circuit is inefficient in that it is many times larger and slower than the corresponding synchronous design. As a result, the FPGA implementations were considered to be useful only for initial functional debug, and to allow initial comparison with synchronous designs that had been set up specifically for FPGA implementation.

To support full ASIC implementation, and to allow closer comparisons with standard Boolean layouts, it has been necessary to develop a specific standard cell library for NCL as there were none available at the time this work was undertaken. These libraries were "height-matched" to the existing Boolean cells in the 28nm FDSOI process used in this work, so that

some of the existing Boolean standard cells could be used where necessary and appropriate. These NCL cell libraries will be used in the following chapters to illustrate the behavior of NCL in arithmetic and processor architectures.

Chapter 4

NCL Circuit Designs for CPU

This chapter introduces the range of CPU components from which the NCL based 32-Bit CPU core (Chapter 5) has been built, as well as their circuit-level optimization techniques. The various arithmetic solutions such as Adders, Shifters and Multipliers also serve as useful illustrative examples to show how these NCL circuits can be designed and optimized. In the first section, a number of different types of NCL Adder designs are introduced and their comparison results discussed. The discussion then moves on to Multiplier circuits. High speed multipliers are a fundamental resource for a broad range of DSP applications such as Finite Impulse Response (FIR) Filters or Fast Fourier Transform (FFT) circuits and also for Flow Graph designs like streaming media processing (Video and Audio), Ethernet Packet processing and Cryptography circuits and so on. Finally, an asynchronous shifter design is included here that is later used in the NCL based RISC-V arithmetic unit.

Following the analysis of the arithmetic circuits, two other important CPU blocks are discussed: the Register File and Program Counter. These blocks are amongst the most complex and challenging to design using NCL. In the case of the Program Counter, it is shown how the concept of the NCL Ring-Oscillator can be used to impart *"liveness"*[1] to the CPU circuit without the use of an external clock source.

4.1 NCL Adder Design

In this section, the NCL adder designs are introduced, in particular the 32-Bit adder structures used for the CPU Arithmetic Logic Unit, plus the Program Counter and Load/Store Unit. Also discussed are the 64-Bit Adders used in the final 32x32 Multiplier architecture. The adder designs are either simple Ripple-Carry Adders (RCA) or Parallel-Prefix Adder (PPA)

[1] See Ch. 2 for a brief overview of "liveness".

types. Using those RCAs and PPAs, the benefits of fine-grained 1- or 2Dimensional pipelining of NCL circuits have been tested. Fine-grained registration and pipelining is one of the benefits of asynchronous design, particularly for NCL. The comparison results between 2Dimensional (2D) Ripple-Carry Adders, Parallel-Prefix Adders with non-pipelined designs are also shown.

4.1.1 Fundamental Theory of the NCL Adder

This section introduces the NCL Full-Adder design and how it is optimized using the NCL fundamental gates [109] [19].

NCL Full Adder Combinational Logic

Just as for Clocked Boolean Logic, dual-rail NCL combinational logic can be optimized using a Karnaugh-Map approach. Table 10 shows the Truth Table of 1-bit Full Adder, which can be simplified to an NCL logic description in terms of its Boolean algebra expressions using a K-map. Table 7 in the previous chapter showed the combinational expression for their Set and Reset networks for each NCL cell and those expression can be used to derive the appropriate terms for the Adder design. Note that in clocked Boolean logic, we only consider the '1' entries on the Karnaugh-Map, but in NCL, both '1' and '0' entries contribute to the NCL dual-rail outputs.

Table 10: Truth Table for Full Adder

X	Y	Ci	Co	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

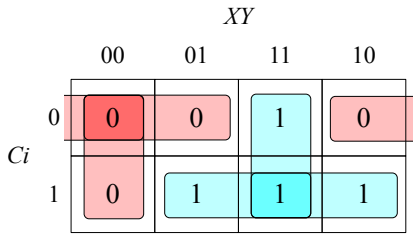

Figure 59: Full Adder Karnaugh Map Simplification for Carry Out

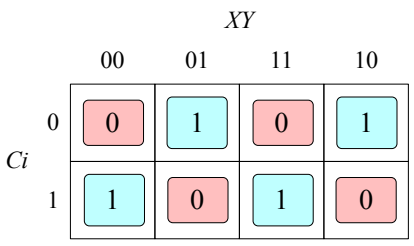

Figure 60: Full Adder Karnaugh Map Simplification for Sum

Figure 59 presents a simplified Full Adder Truth Table for dual-rail *Carry Out* (Co). Both outputs (Equation 4.1) can be directly mapped to a TH23 NCL gate based on Table 7 Line Number 4.

$$\begin{aligned} Co^0 &= X^0Y^0 + Ci^0X^0 + Ci^0Y^0 \\ Co^1 &= X^1Y^1 + Ci^1X^1 + Ci^1Y^1 \end{aligned} \tag{4.1}$$

Dual-Rail *Sum* output has no simplification (Figure 60) and therefore the equations become:

$$\begin{aligned} S^0 &= X^0Y^0Ci^0 + X^0Y^1Ci^1 + X^1Y^0Ci^1 + X^1Y^1Ci^0 \\ S^1 &= X^0Y^0Ci^1 + X^0Y^1Ci^0 + X^1Y^0Ci^0 + X^1Y^1Ci^1 \end{aligned} \tag{4.2}$$

These equations can be simply expressed using four TH33 NCL gates (equivalent to a Boolean AND gate) and one TH14 gate (equivalent to an OR gate) without simplification.

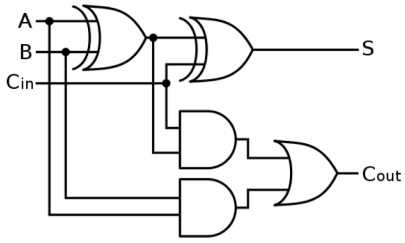

Figure 61: Full Adder Design using Two Half Adders

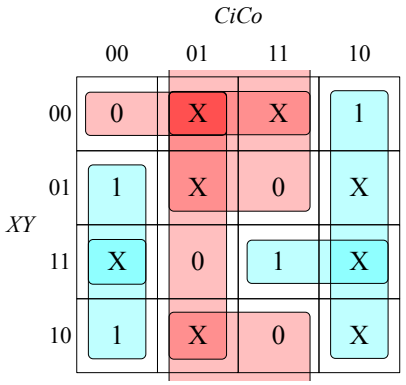

Figure 62: Full Adder Karnaugh Map Simplification for Sum with Co Inputs

The Full Adder equations also can be expressed as a composition of two Half Adders. Figure 61 and Figure 63 display this version of the circuit. To reduce the number of gates without increasing delay, we can use *Co* output as an input signal of the *Sum* equations and re-use the truth table from Table 10.

These results can be further simplified using the K-map of Figure 62. This K-map now has four inputs - *Ci, Co, X and Y* based on the Table 10, with *Co* becoming an input of the K-map for the optimized Sum output generation). The following are the simplified equations for Sum, both of which directly map to a TH34W2 gate.

$$\begin{aligned} S^0 &= Co^1X^0 + Co^1Y^0 + Co^1Ci^0 + X^0Y^0Ci^0 \\ S^1 &= Co^0X^1 + Co^0Y^1 + Co^0Ci^1 + X^1Y^1Ci^1 \end{aligned} \tag{4.3}$$

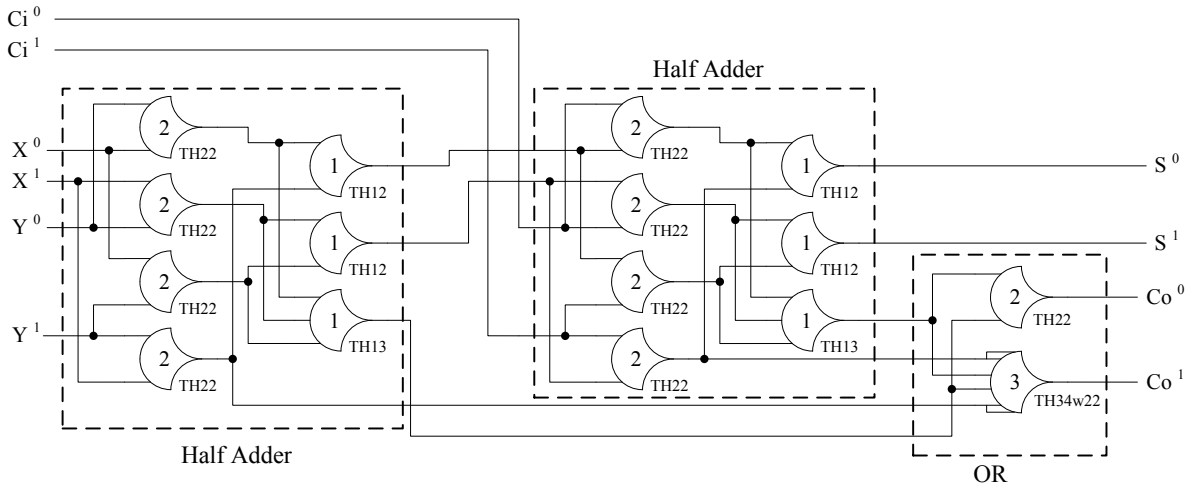

Figure 63: 1Bit Full Adder Circuit by using Two Input NCL Gates

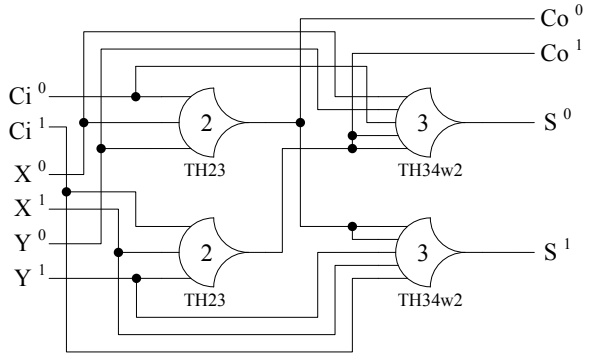

Figure 64: Final Optimization Results of 1Bit Full Adder - redrawn from [18] [19]

Finally, we can obtain the optimized result (Figure 64). This uses two TH23 gates and two TH34W2 gates [18] [19]. Compared to the original Full Adder designed just using 2-input NCL gates (Figure 63) the area of the circuit has been significantly reduced.

4.1.2 Introduction to NCL Adders

Initially, all the possible clocked Boolean adder types were considered for NCL Adder design. Potentially, all adder types can be converted to their NCL equivalents but some can be done so much more efficiently than others. Table 11 shows a list of adder styles that were considered. There are two major classes of Binary adder: Ripple-Carry Adder (RCA) and Parallel-Prefix Adder (PPA). In synchronous designs the RCA is hardly ever used for real CPU architectures because of its worst-case carry-chain delay. However, in NCL, the RCA exhibits an average-case carry-chain delay and is therefore widely used for arithmetic designs as it can significantly reduce the area overheads.

Binary Adder Types

Table 11 shows the ten Binary Adder types in common use in clocked Boolean circuits.

Table 11: Binary Arithmetic Adder Types

No	Adder Type	Specification
1	Ripple-Carry Adder	Each carry bits rippled into the next stage , smaller but slower
2	Carry Look-ahead Adder	Calculates one or more carry bits before the sum, and dividing the adder into blocks
3	Kogge-Stone Adder	PPA, Long wires, More PG cells, Widely used
4	Brent-Kung Adder	PPA, Less wires, More stages
5	Han-Carlson Adder	PPA, Mix of Kogge-Stone and Brent-Kung, Trades logical level for wire length
6	Ladner-Fischer Adder	PPA, Combining Brent-Kung and Sklansky
7	Carry Select Adder	Generally consists of two ripple carry adders and a multiplexer, one time with the assumption of the carry-in being zero and the other assuming it will be one, simple but rather fast
8	Condition Sum Adder	Generate two sets of outputs, select the correct set of outputs by incoming Carry
9	Carry-Skip Adder	Using the Carry-Skip logic, reduced worst-case delay, using several carry-skip adders to form a block-carry-skip adder
10	Carry-Save Adder	Output has two separated formats - Sum and Carry, result still has to be converted back into binary (Final Adder) It has advantages when sum of three or more numbers

(1) The Ripple-Carry is the most basic type of adder. Figure 65 shows a 4-Bit RCA illustrating that each Carry bit ripples to the adjacent stage. While it is the smallest adder, its worst-case carry-chain propagation delay is proportional to its bit width.

(2) The Carry Look-ahead Adder computes one or more carry bits before the sum, which reduces the wait time to calculate the result of the larger-value bits of the adder and divides the adder into multiple blocks to reduce the carry-chain delay [110]. Kogge-Stone, Brent-Kung, Han-Carlson, Ladner-Fischer Adders are most well known of Parallel Prefix Adder types.

(3) The Kogge-Stone Adder [111] is the most widely used but requires more *Propagate* and *Generate* cells and exhibits higher power consumption than other PPAs. Figure 66 shows the 8-Bit Kogge-Stone Adder example (*GPn*(blue): Generation and Propagation, *BCn*(black): Black Prefix Cell, *GCn*(grey): Grey Prefix Cell, *Sumn*(green): Final XOR, *Bn*(yellow): Buffer).

(4) The Brent-Kung Adder [112] avoids the explosion of wires that characterize other PPAs but has more stages than Kogge-Stone.

(5) Han-Carlson Adder [113] is a mix of Kogge-Stone and Brent-Kung. It trades logic depth for

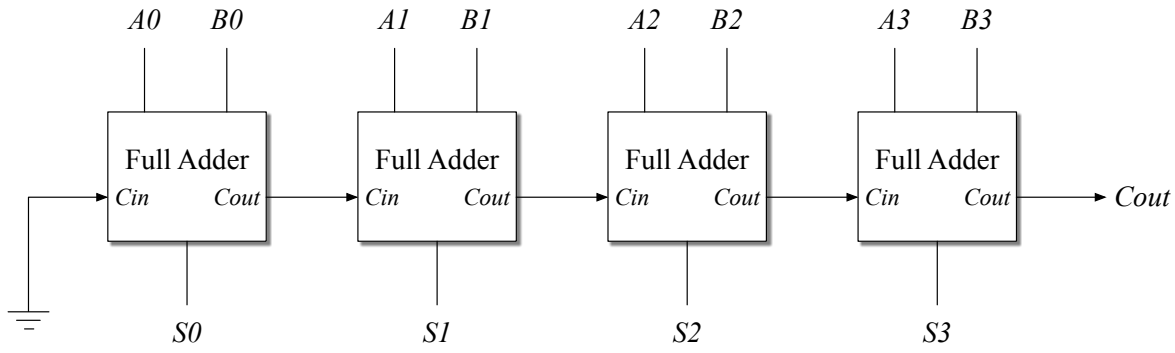

Figure 65: Boolean 4-Bit Ripple Carry Adder

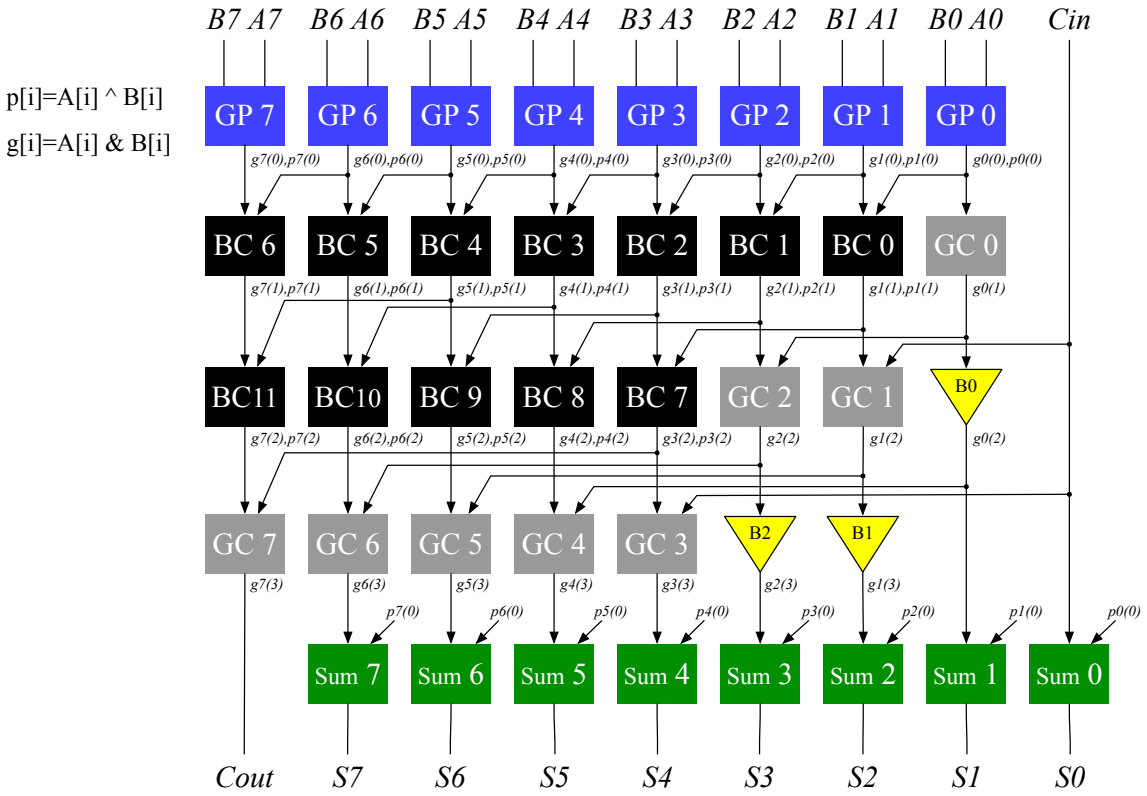

Figure 66: 8-Bit Boolean Kogge-Stone Adder

wire length.

(6) The Ladner-Fischer Adder [114] has a minimum logic depth and large fan-out requirement—-up to n/2 for a n-bit addition.

(7) The Carry Select Adder [115] generally uses two Ripple-Carry Adders and Multiplexers such that the "correct" RCA value is selected by the carry-in.

(8) The Condition Sum Adder [116] is a recursive structure based on the carry-select adder. In

the same way as the CSA, it has two sets of adders and the output is selected using the incoming carry input signal. The adder bit width is divided into smaller groups to avoid carry propagation. The outputs of these subgroups are then combined to generate the output of the groups.

(9) The Carry-Skip Adder [117] is implemented using Block-carry-skip adders. The block size and the optimization level need to be determined to optimize this adder.

(10) The Carry-Save Adder [118] usually uses a tree of multipliers and has two separate outputs, Sum and Carry. These still need to undergo a final addition to achieve the binary result. The CSA exhibits advantages when the sum of three or more numbers needs to be computed.

The NCL Ripple Carry Adder

Figure 67 shows an example of a 4-Bit dual-rail NCL Ripple Carry Adder. The optimized 1-Bit Full-Adder of Figure 64 is used for this RCA and just as for the equivalent Clocked Boolean RCA, the carry bits are rippled to the adjacent Full-Adder.

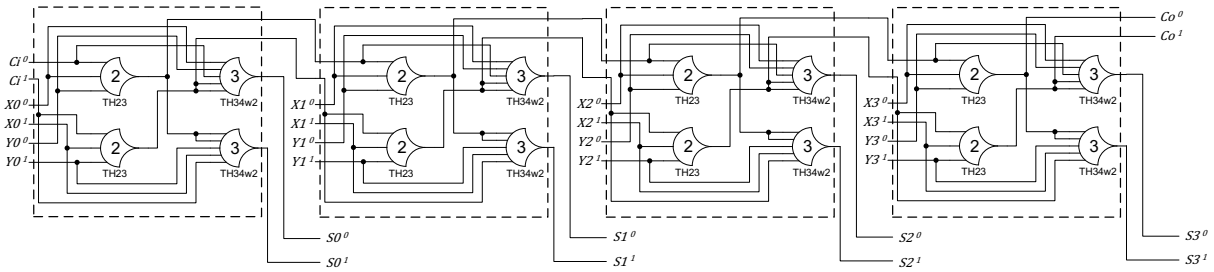

Figure 67: 4-Bit NCL Ripple Carry Adder

Parallel Prefix Adder

The Parallel Prefix Adder (PPA) is a high speed Adder design without Carry propagation and the addition operation is processed in parallel. The Parallel Prefix Adder has a 3-stage structure: (1) Pre-Calculation of Propagation (*Pi*) and Generation (*Gi*); (2) Calculation of the Carries (*Ci*) and (3) Combination of Carry and Propagation and generation of the output Sum (*Si*). The addition equations are factored into Generate (G) and Propagate (P) terms as:

$$\begin{aligned} P_i &= A_i \oplus B_i \\ G_i &= A_i \bullet B_i \end{aligned} \tag{4.4}$$

Calculation of Carry and Sum then becomes:

$$\begin{aligned} G_{i:j} &= G_{i:k} + P_{i:k} \bullet G_{k-1:j} \\ P_{i:j} &= P_{i:k} \bullet P_{k-1:j} \\ Si &= P_i \oplus G_{i-1:0} \end{aligned} \tag{4.5}$$

Parallel Prefix Adders compute Carry at each level of addition by combining the Propagate and Generate terms. The formulation of the NCL adder design is identical to that of the syn-

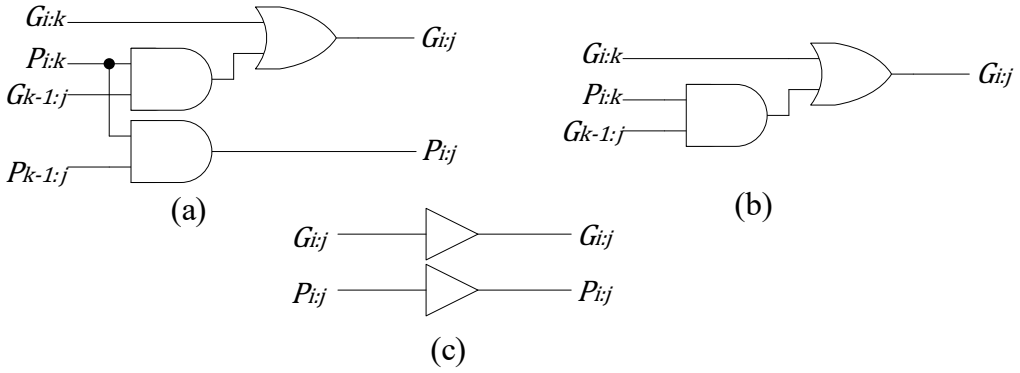

Figure 68: Boolean PPA Prefix Cells

chronous case. Figure 68 shows the details. There are three types of cells shown here making up the PPA: the Prefix Cells (a)–shown in Black–that generate one set each of the Propagate and Generate signals, the Gray Prefix Cells (b) that derive only the Generate signal (i.e., no propagate), and Buffers (c).

NCL Parallel Prefix Adders

For the NCL based Adder circuits, the Kogge-Stone and Han-Carlson Parallel Prefix Adder were selected due to their widespread use in computer architectures. Figure 69 shows a 64-Bit NCL based Kogge-Stone Adder and Figure 70 the 64-Bit NCL based Han-Carlson Adder. The cells in these diagrams are the same as in Figure 66 but in these cases, all the cells are dual-rail. The Prefix Cells of both adder types are based on simple dual-rail NCL **AND** and **OR** gates such as in Figure 71 and 72. These gates were then optimized to such as is shown in Figures 73 and 74. As for clocked Boolean prefix cells (Figure 68), NCL circuits have two gate delays for Black and Grey Prefix Cells. However, NCL has double the cell count because of its dual-rail nature.

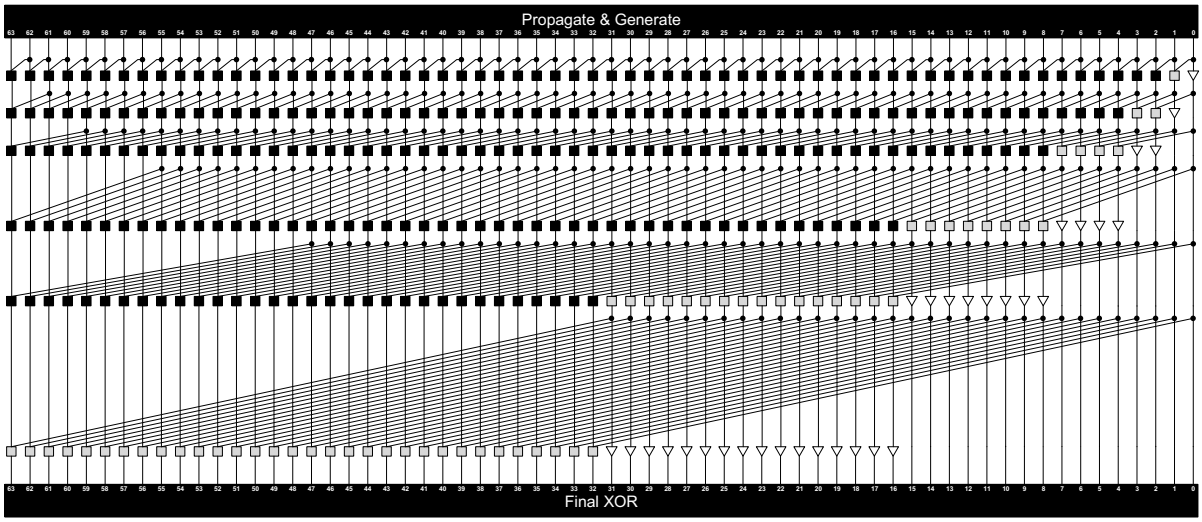

Figure 69: 64-Bit NCL Kogge-Stone Adder

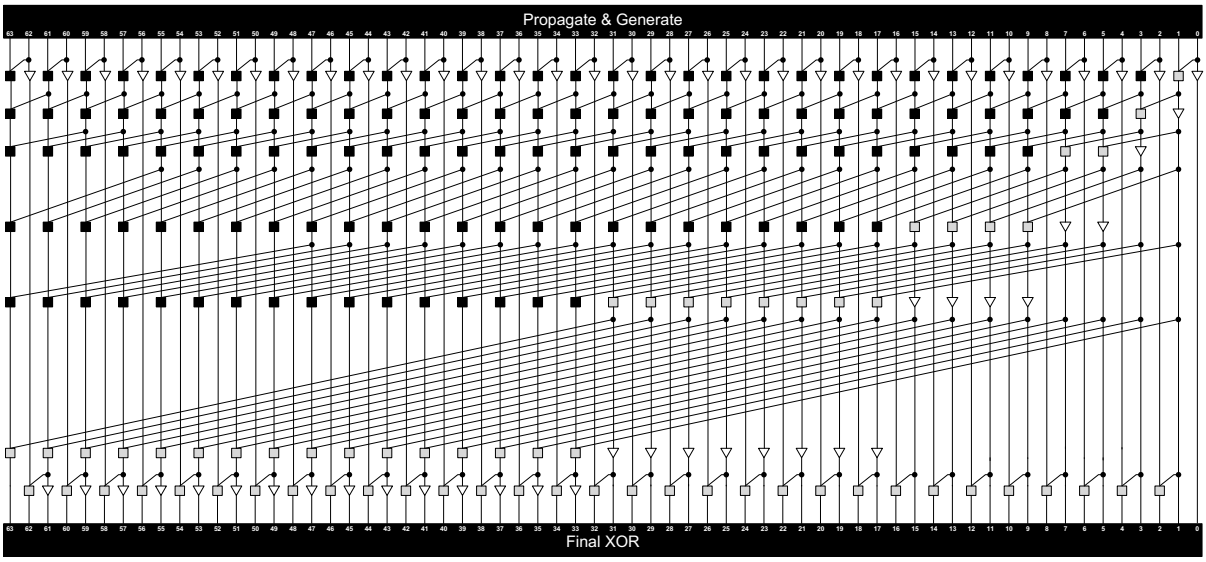

Figure 70: 64-Bit NCL Han-Carlson Adder

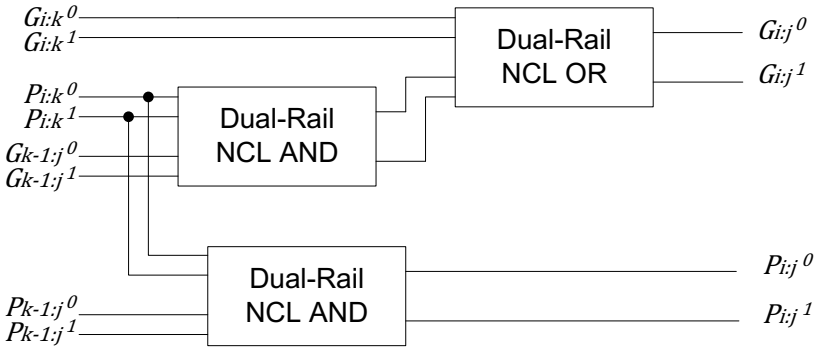

Figure 71: PPA Carry Operator - Black Cell

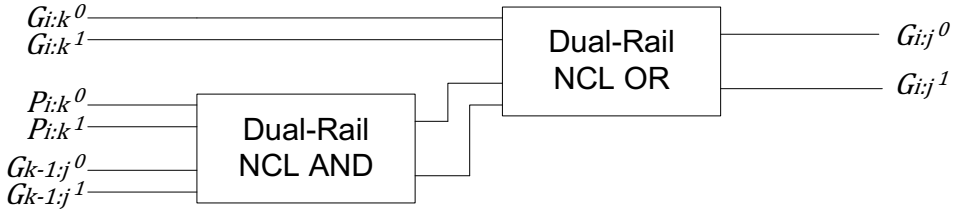

Figure 72: PPA Carry Operator - Grey Cell

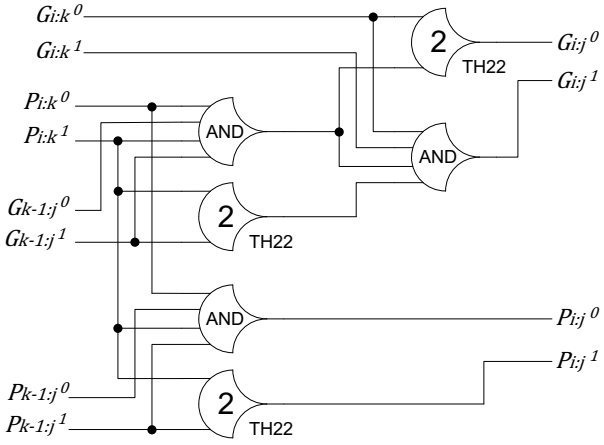

Figure 73: Optimized NCL Carry Operator - Black Cell

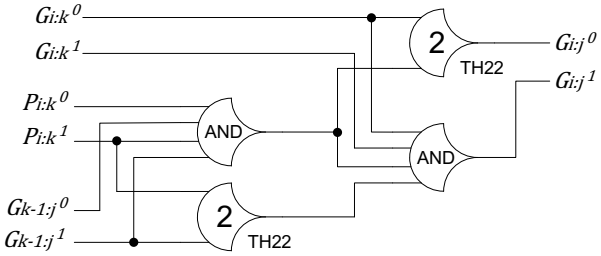

Figure 74: Optimized NCL Carry Operator - Grey Cell

4.1.3 Two-Dimensional Pipelined NCL Adders

NCL circuits can be optimized for high performance by integrating register functions with combinational logic to reduce overall delay [18]. The natural pipelining behavior of NCL can often result in high speed data paths with fewer gate delays. However, the spanning completion detection and shared completeness path of the NCL handshaking signal becomes a major concern for larger circuits using 1Dimensional (1D) pipelining as it can result in the need for very large completion detection gates that exhibit excessive fanin, long propagation delays and high capacitance. Fine grained 2D pipelining is a potential solution to this problem.

One of the key advantages of NCL is that it is relatively easy to integrate registers into the combinational logic paths. Thus, the various combinational components (full and half adder, dual-rail AND gate, Black Cell and Grey Cell) have been treated in this way and before being connected to their adjacent gates. The major advantages here are a reduction in area and overall complexity, plus a significant increase in throughput due to much shorter critical paths. The following presents the detailed structure of the various components and shows how they can be optimized for efficient 2D pipelining.

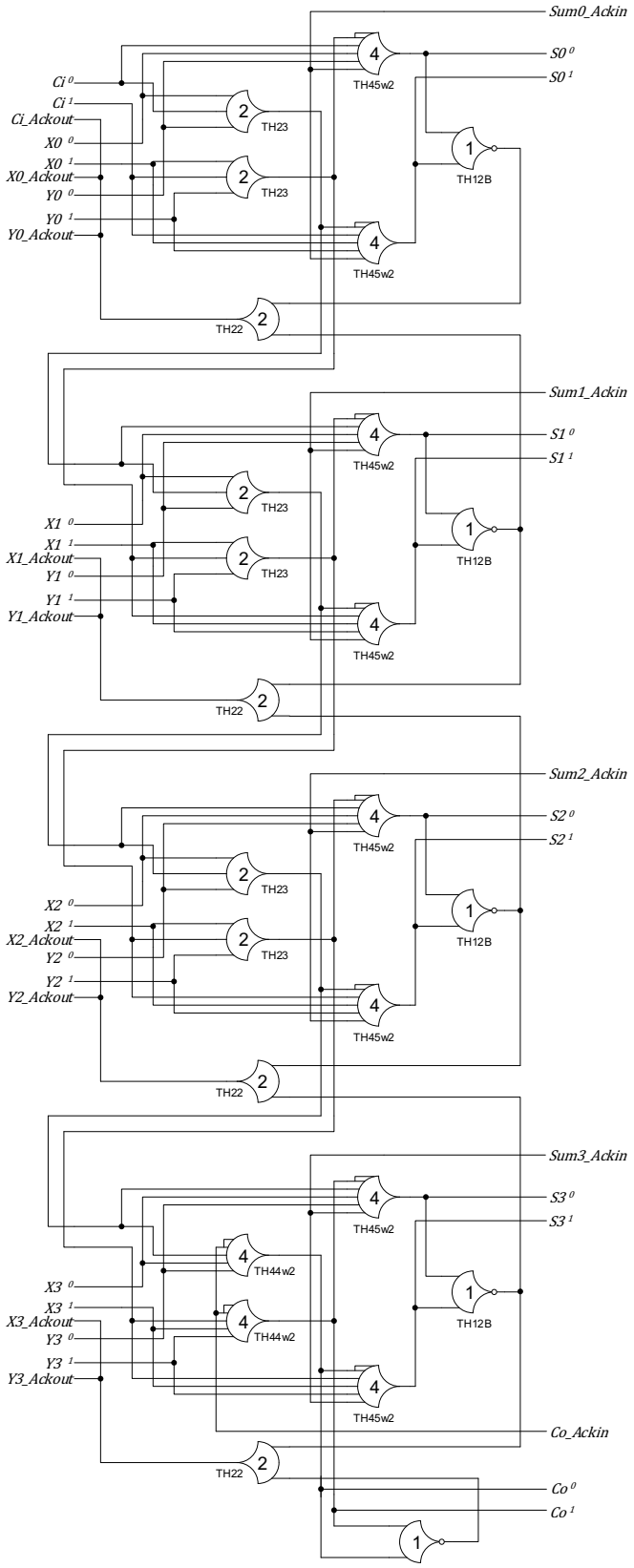

Figure 75: Ripple Carry Adder Dual-Rail NCL with Integrated Registers Carry Chain Propagation (1.5D)

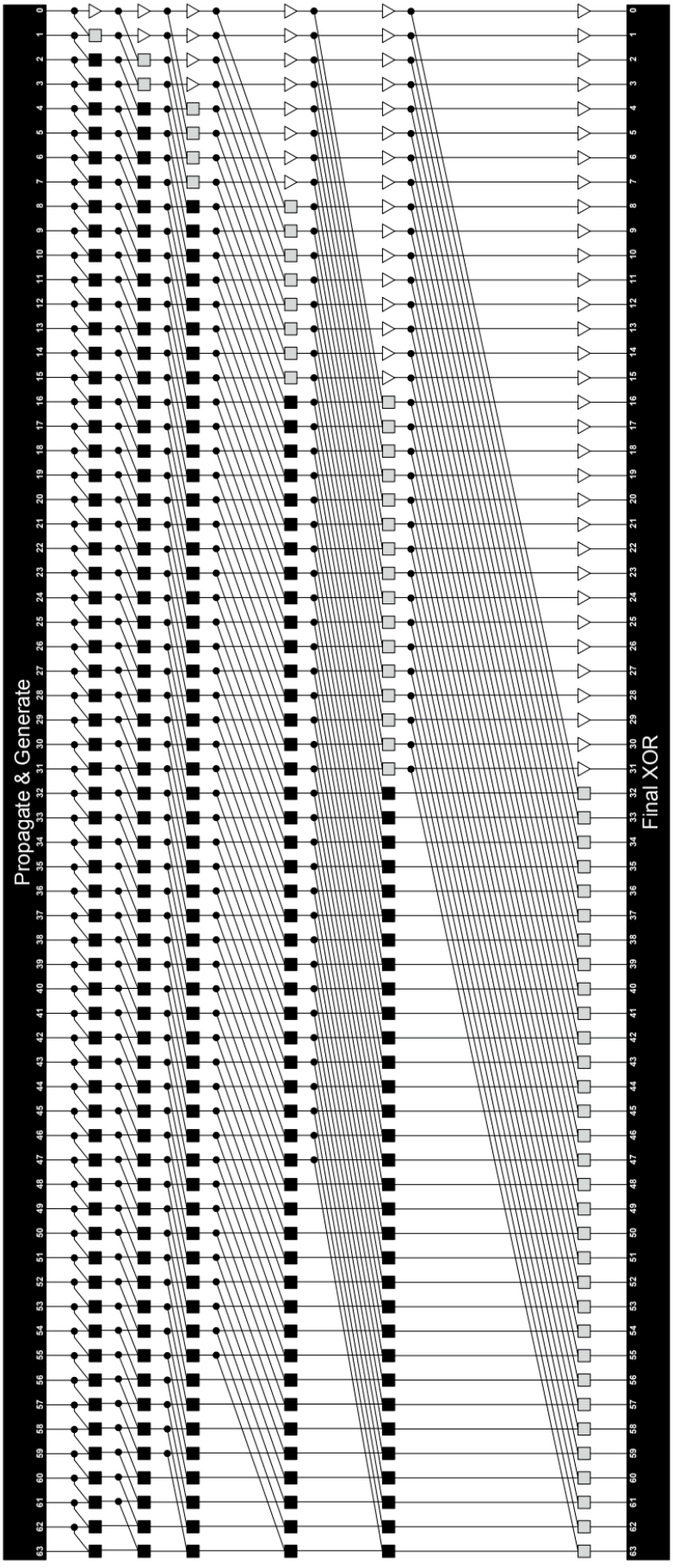

Figure 76: 2D 64-Bit Kogge Stone Adder

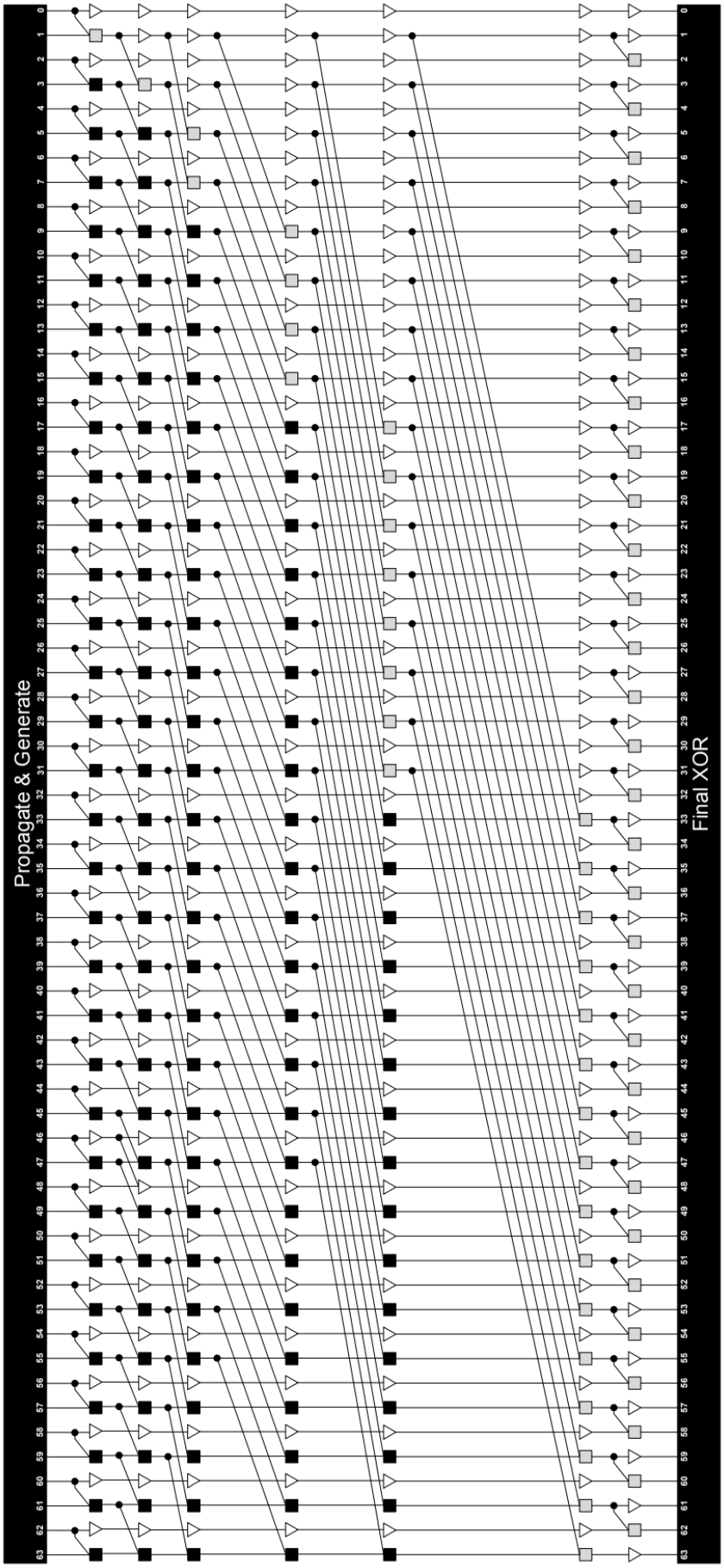

Figure 77: 2D 64-Bit Han Carlson Adder

Figure 75 is a Dual-Rail NCL Ripple Carry Adder with Integrated Registers. It is shown as a 4-Bit adder but can be easily extended to 32 or 64 bits. This adder exhibits average carry-chain delay and each carry chain stage exhibits one NCL threshold gate delay. Figure 75 is not the full 2D design but incorporates fine-grained completion logic including the carry chain. This is termed a '1.5D' Ripple-Carry Adder design.

Figures 76 and 77 show the 2D 64-Bit Kogge-Stone and Han-Carlson adders, respectively. Unlike Figures 69 and 70, these adders have additional buffers to generate equal pipeline stages and each of the cells on the adder include integrated registers and handshaking signals are connected to their previous and next stage registers.

The 2D Kogge-Stone Adder has eight pipeline stages and the 2D Han-Carlson Adder has nine pipeline stages. Figure 78 shows the structure of the Prefix Adder Black Cells for dual-rail 2D pipelined Parallel Prefix Adder. Similarly, Figure 79 shows the optimized Prefix Adder Black Cell with integrated registers. Output Registers and also input/output handshaking signals are included in this design.

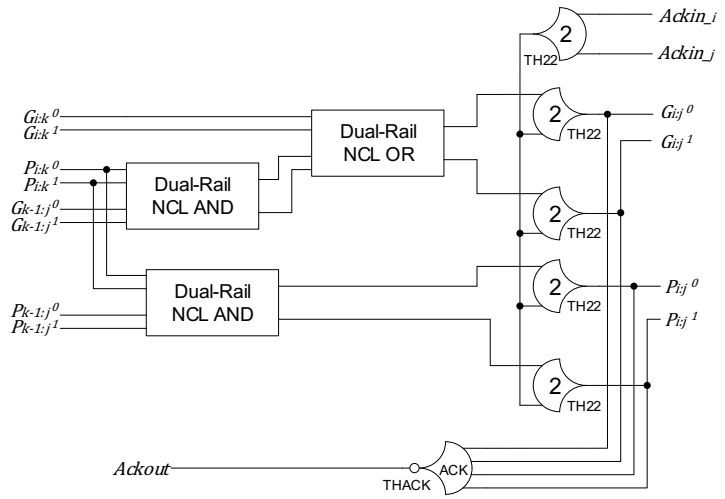

Figure 78: Prefix Adder Black Cell Dual-Rail

Figure 80 shows the structure of the Prefix Adder Grey Cells and Figure 81 the optimized Prefix Adder Grey Cell with integrated registers. These optimized cells are used for both 2D Kogge-Stone and Han-Carlson adders. Buffers for 2D design only have dual-rail registers just to repeat the signals to create balanced 2D pipeline structures.

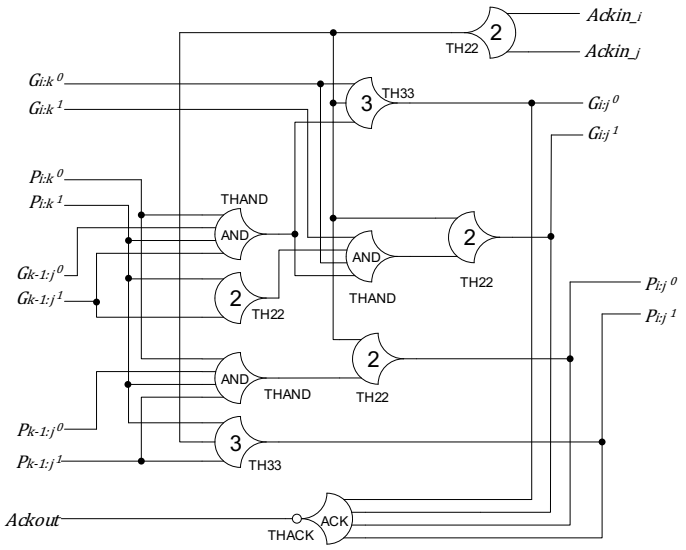

Figure 79: Optimized Prefix Adder Black Cell Dual-Rail NCL with Integrated Registers

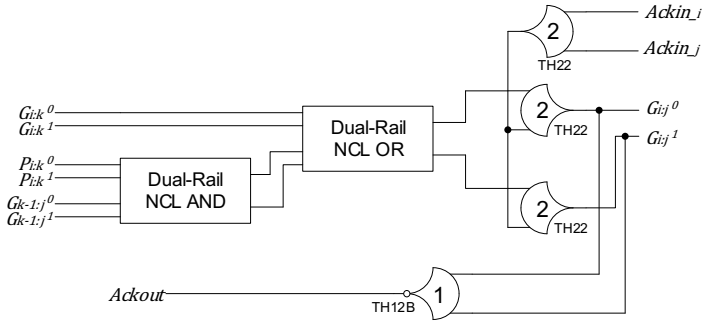

Figure 80: Prefix Adder Gray Cell Dual-Rail

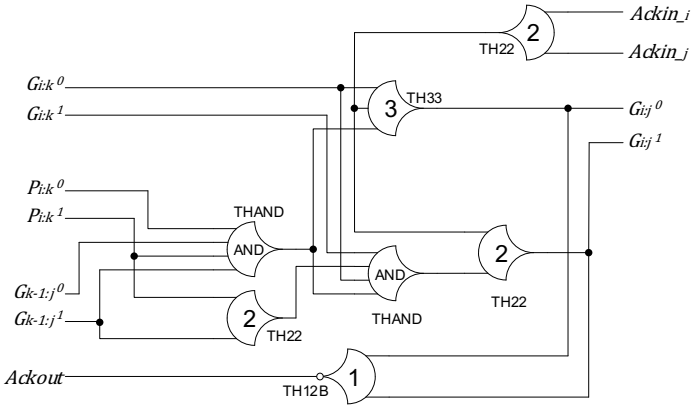

Figure 81: Optimized Prefix Adder Gray Cell Dual-Rail NCL with Integrated Registers

4.1.4 Adder Comparison Results

Table 12 shows the performance and area comparison results of the 64+64 adders. These results were obtained using models for a 28nm FDSOI process. The table includes three

different adders (Ripple-Carry, Kogge-Stone, Han-Carlson) and two different adder architectures (Non-Pipelining, 2D pipelining). Non-pipelining RCA uses the least area but has fastest performance. The Non-pipelined Han-Carlson adder uses less area compared to the Non-pipelined Kogge-Stone adder but, in the 2D architecture, the area and performance are quite similar because of the added Dual-Rail registers in the Han-Carlson Adder.

Table 12: 64-Bit Adder Comparison Results

64+64 Adder	Throughput and Latency Comparison Results			Area Comparison Results
	Cycle-Time	Throughput	Pipeline Depth	NCL Gates
Ripple-Carry Adder(Non-Pipelining)	2,140 ps	467 MHz	1	461
Kogge-Stone Adder(Non-Pipelining)	2,357 ps	424 MHz	1	2,351
Han-Carlson Adder(Non-pipelined)	2,584 ps	386 MHz	1	1,577
Ripple-Carry Adder(1.5D pipelined)	1,592 ps	628 MHz	1	637
Kogge-Stone Adder(2D pipelined)	816 ps	1,225MHz	8	5,890
Han-Carlson Adder(2D Pipelined)	798 ps	1,253 MHz	9	5,885

4.2 NCL Multiplier Design

Multipliers are a fundamental resource for a broad range of applications, including CPUs. In this section, some NCL multiplier designs are analysed, in particular 32-Bit industry standard high speed multipliers intended to support CPU design. Initially, the fundamentals of NCL multiplier architectures are introduced using the basic Array Multiplier as an example. Then, the high speed Modified-Booth multiplier and Baugh-Wooley Multiplier are described. We also show a Multiplier based on ancient Indian Vedic Mathematics. As for the previous adder case, 2Dimensional Multipliers are described. Using the simple 4-Bit Array Multiplier, the impact of pipelining on the Multiplier performance is analysed in detail. Finally, 2D Baugh-Wooley Multipliers are analysed and their comparative results shown.

4.2.1 Fundamental Theory of NCL Multiplier

Multiplication structures can be organized generally into serial or parallel classes. Serial Multiplication (Shift and Add) computes a set of partial products, which are then summed together. The technique is used when there is a lack of dedicated multiplier hardware. Parallel multiplication produces the partial products simultaneously and adds them with a final

high performance addition stage. The technique is used when computational performance is important.

Equation 4.6 shows the multiplication of operand X (m-bit integer) and Y (n-bit integer) to result in product P(m + n bit integer). This equation can be expressed in terms of the fundamental NCL gates as:

$$X = \sum_{i=0}^{m} x_i 2^i \quad and \quad Y = \sum_{j=0}^{n} y_j 2^j$$
$$P = XY = \left(\sum_{i=0}^{m} x_i 2^i\right)\left(\sum_{j=0}^{n} y_j 2^j\right) = \sum_{i=0}^{m} 2^i \left(\sum_{j=0}^{n} x_i y_j 2^j\right) = \sum_{i=0}^{m}\sum_{j=0}^{n} x_i y_j 2^i 2^j \tag{4.6}$$

In NCL, the dual-rail NCL AND (Figure 83) is used for Partial Product generation and the dual-Rail NCL full-adder (Figure 64) is used as the basic addition element. Figure 82, 83 and Equation 4.7 show how the dual-rail NCL AND function is optimized using the fundamental NCL gates. Output Z_0 is the combination of input $A_0B_0 + B_0A_1 + A_0B_1$ and is matched with THAND gate of Table 7, line 26 [2]. Output Z_1 is matched with a TH22 gate.

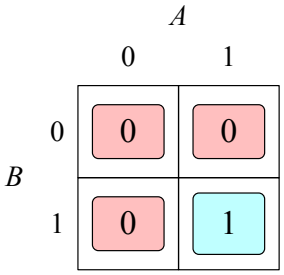

Figure 82: Dual-Rail NCL AND Karnaugh Map

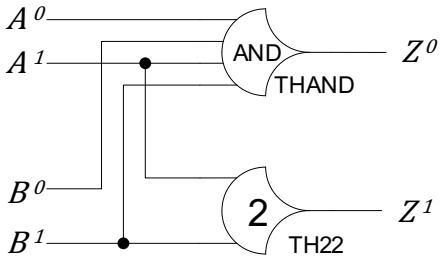

Figure 83: Dual-Rail NCL AND

$$Z^0 = A^0B^0 + B^0A^1 + A^0B^1$$
$$Z^1 = A^1B^1 \tag{4.7}$$

4.2.2 Introduction to NCL Multipliers

As was previously done for the NCL Adder designs, all the possible Clocked Boolean Multiplier types were also considered as potential candidates for the NCL Multiplier designs. Table 13 shows a list of multiplier styles that were considered.

[2] This is reason why the gate is named the THAND

Binary Multiplier Types

(1) Array Multipliers are typically organized as a combinational array of Full- and/or Half Adders. Figure 84 shows an example 4 x 4 Multiplier formed from only Full Adder components in which some of the inputs have been set to zero (effectively creating half adders). The Partial Products $X_i \bullet Y_j$ can be summed in the conventional manner such as using a matrix of Ripple-Carry Adders.

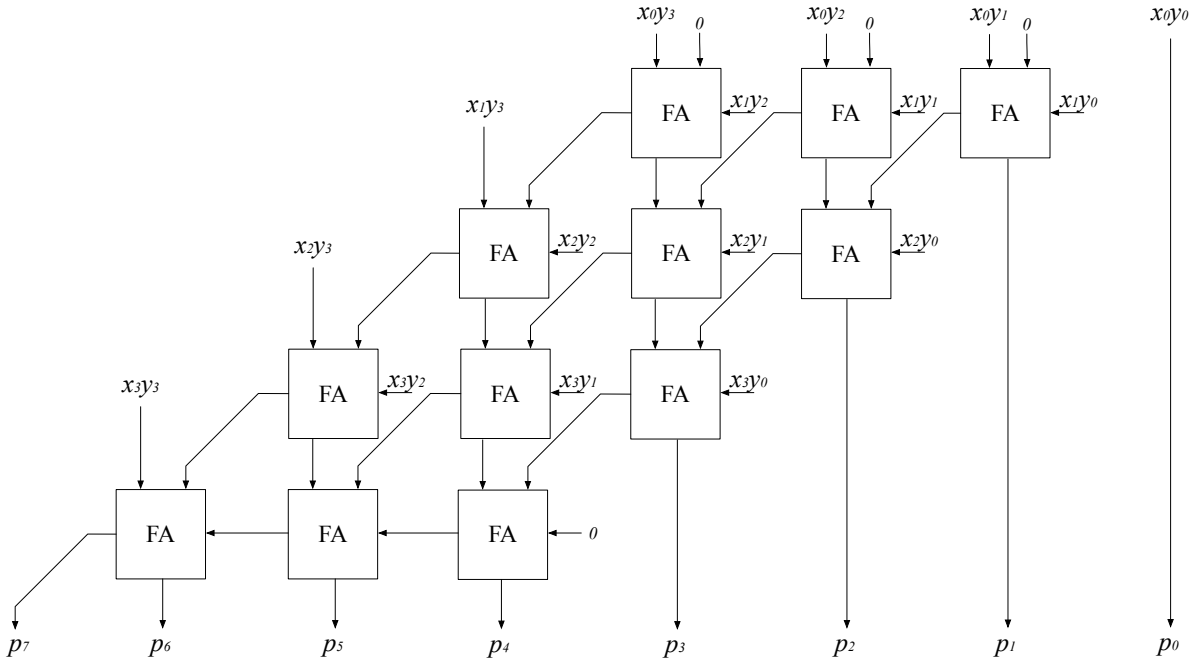

Figure 84: 4-Bit Array Multiplier

(2) The Baugh-Wooley Multiplier [119] is the most well known signed multiplier, because it maximizes the regularity of the structure and allows all of the partial products to have positive sign bits. This multiplier generates all the possible partial-products first using AND terms and then sends these through the adder array with the carry chain to the next most significant bit at each level.

(3) The Vedic Multiplier [120] - Vedic Mathematics is an ancient form of mathematics reconstructed from ancient Indian scriptures referred to as *Vedas*. Vedic Mathematics is based on the natural principles on which the human mind works [121]. It was really interesting to see whether ancient Vedic Mathematics fits the data-flow style NCL designs.

(4) The Wallace-Tree Multiplier [122] is the most widely known tree multiplier architecture. The tree architecture is used to reduce the partial products using Carry-Save Adders(Compressors)

and reduces both the critical path delay and the number of adder cells. The output of this architecture has Carry-Save format (Sum and Carry) therefore it needs one further final addition step.

(5) The Booth Multiplier is the most well known multiplier used in CPU design. The multiplier takes two signed binary numbers in two's complement notation as inputs. The algorithm reduces the Partial-Product count by using a different radix. For example, in radix-4, after comparing the input bit(B_i) with two adjacent bits(B_{i+1}, B_{i-1}), the Partial-Product count is halved.

Table 13: Binary Arithmetic Multiplier Types

No	Multiplier Type	Specification
1	Array Multiplier	Basic Parallel Multiplier, combinational array of Full-Adder structure.
2	Baugh Wooley Multiplier	most well known signed multiplier, maximized the regularity of the multiplier
3	Vedic Multiplier	Based on ancient Indian Vedic Mathematics, expand the width with the hierarchical structure
4	Wallace-Tree Multiplier	Widely known tree multiplier architecture, using Carry-Save Adder implemented faster and smaller multiplier
5	Booth Multiplier	Reduced Partial-Product count using Booth Algorithm

NCL Vedic Multiplier

As mentioned previously, the Vedic multiplier arises from the ancient Indian Vedic mathematics and is said to be based *"on the natural principles on which the human mind works"* [121]. The Vedic Multiplier exhibits a hierarchical architecture built up from a 2-bit module. Figure 85 shows the clocked Boolean 2-Bit Vedic multiplier architecture. As Figure 85 shows, a 2-Bit Vedic multiplier needs four 2-input AND gates (partial-product generation) and two half-adders. This remains the same for the NCL implementation (Figure 86), which uses four Dual-Rail NCL AND modules (Figure 83) and two dual-rail NCL half-adders (Figure 87). The dual-rail NCL half-adder is optimized via a Karnaugh Map in the same way as the dual-rail NCL full-adder (Figure 64).

Figure 88 shows the 4-Bit Vedic multiplier using four of the 2-Bit multiplier blocks and three adder blocks of varying widths. The size can easily be expanded to 8-Bit, 16-Bit and/or 32-Bit as shown in Figure 89. The NCL Vedic multiplier was implemented in the same way as the Boolean case but with a dual-rail structure.

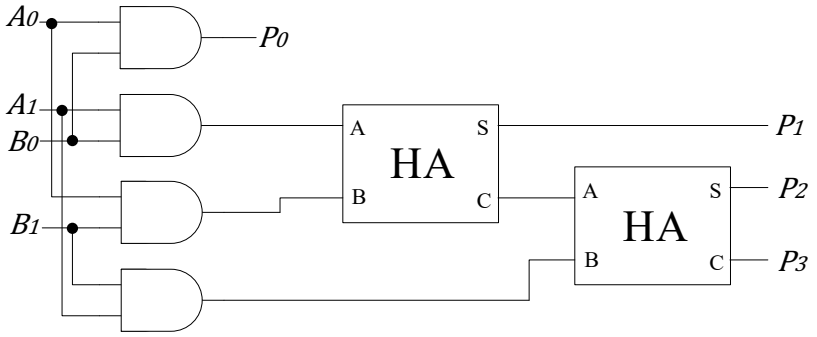

Figure 85: 2-Bit Vedic Multiplier

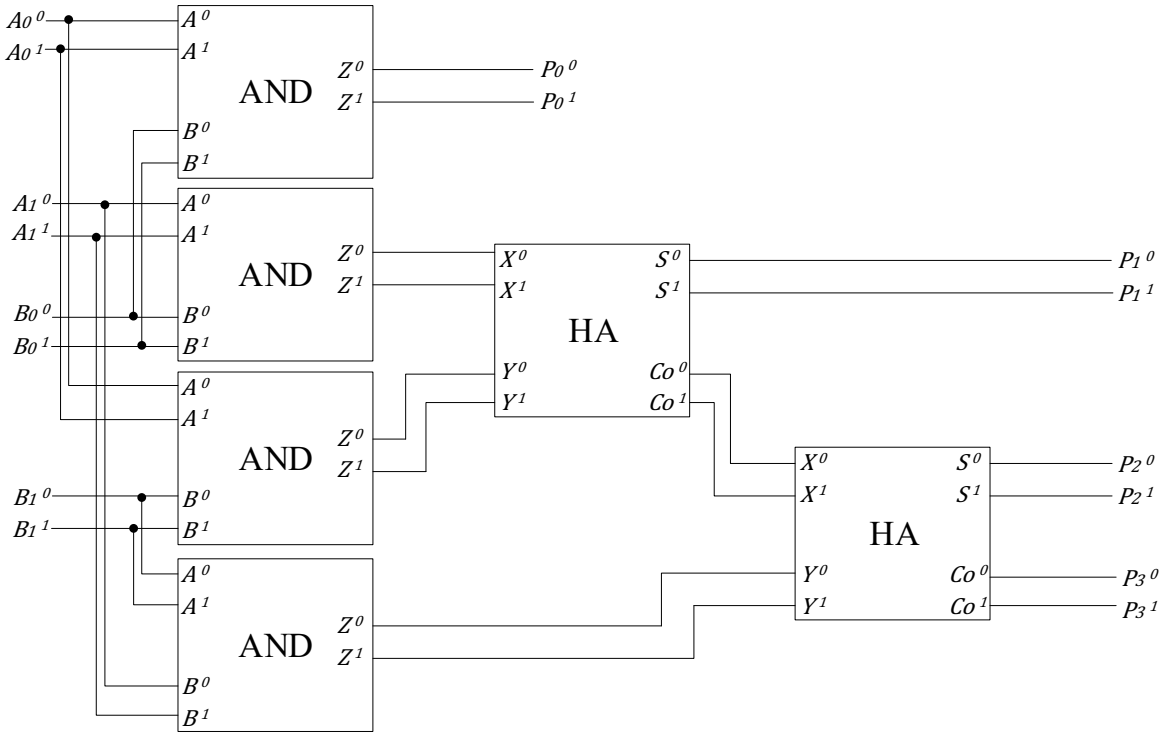

Figure 86: 2-Bit NCL Vedic Multiplier

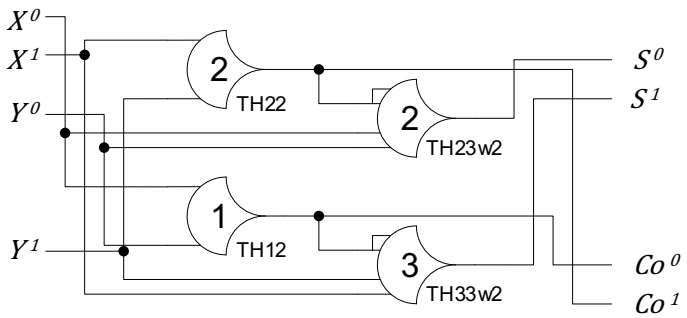

Figure 87: NCL Half Adder

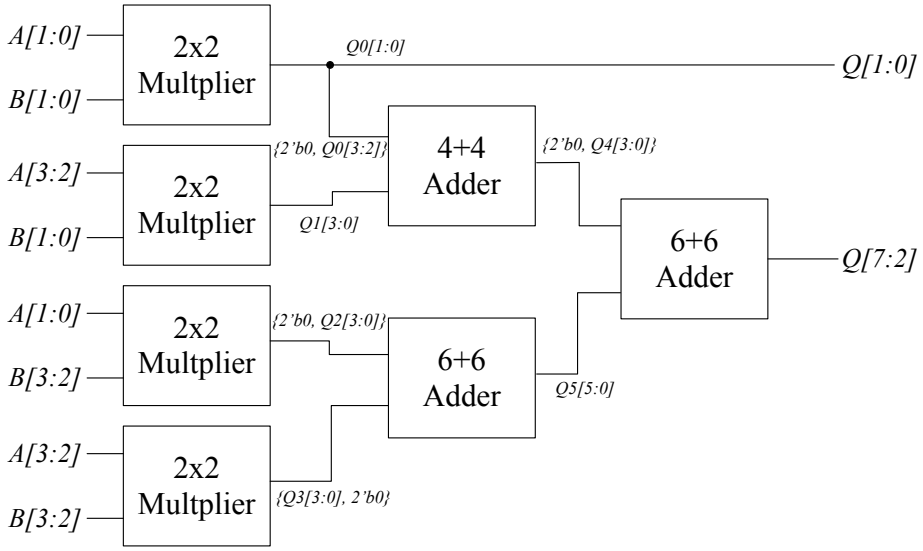

Figure 88: 4-Bit Vedic Multiplier Structure

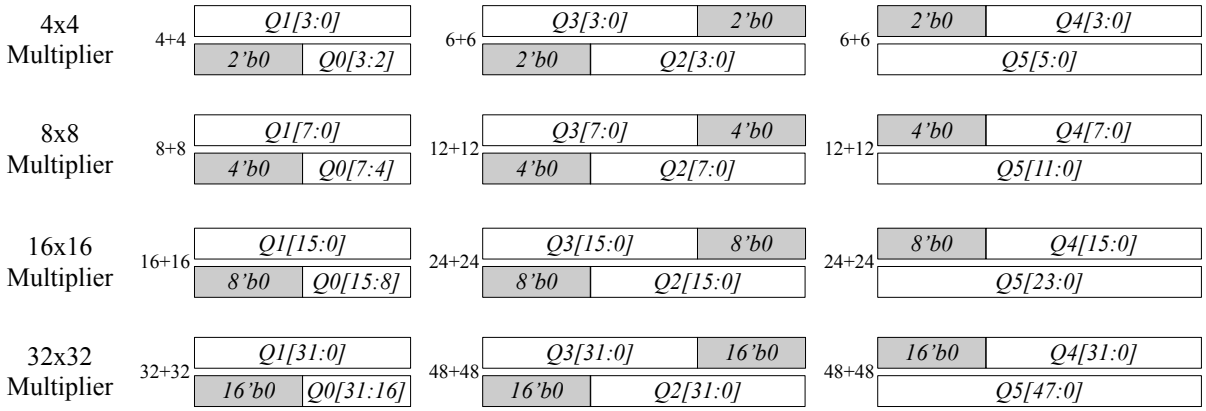

Figure 89: 32-Bit Vedic Multiplier Addition

NCL Modified Booth Multiplier

The Booth Algorithm, introduced by Booth in 1950 [123] is a parallel multiplier method comprising three major parts: (1) Partial Product Generation, (2) Partial Production Reduction and (3) Final Carry Propagation Addition. The Booth algorithm focuses on the first part in order to generate the partial products more efficiently, thereby reducing the area and increasing performance. There are several radix types applied in the Booth Algorithm: radix-2, radix-4, radix-8 and radix-16 etc. Radix-2 can have an advantage where the multiplier data contains continuous sequences of '1's, but otherwise is not very efficient at reducing the Partial-Product count. To solve this issue, encoding with 3-Bits is suggested, resulting in the Radix-4 modified-Booth algorithm. Compared to larger radix values such as radix-8 and 16, radix-4 results in a simple

implementation using only 1-bit shifting and can implement encoding without requiring addition within the partial-product generation circuits. In this work, only radix-4 Modified Booth Multiplier organizations have been implemented.

Taking X as the multiplicand and Y as the multiplier (P: Product), the equation for signed multiplication can be stated as:

$$\begin{aligned} X &= -x_{m-1}2^{m-1} + \sum_{i=0}^{m-2} x_i 2^i \\ Y &= -y_{n-1}2^{n-1} + \sum_{j=0}^{n-2} y_j 2^j \\ P &= XY \end{aligned} \tag{4.8}$$

When the multiplier is reduced using the radix-4 Booth encoder, Y can be expressed as:

$$Y = \sum_{j=0}^{n/2-1} (-2y_{2j+1} + y_{2j} + y_{2j-1})2^{2j} \tag{4.9}$$

In radix-4, after comparing the input bit (b_i) with two adjacent bits (b_{i+1}, b_{i-1}), the partial product count is reduced to half its former value. Figure 90 shows the 32-Bit radix-4 encoding, (b_{i+1}, b_i, b_{i-1}) compared with its new encoding ($B_{16} : B_0$). Note that this is a signed implementation and thus needs 17 partial products. To complete the 3-Bit combinations, two zero bits were added after b31. Table 14 shows the relationship between the multiplier bits and

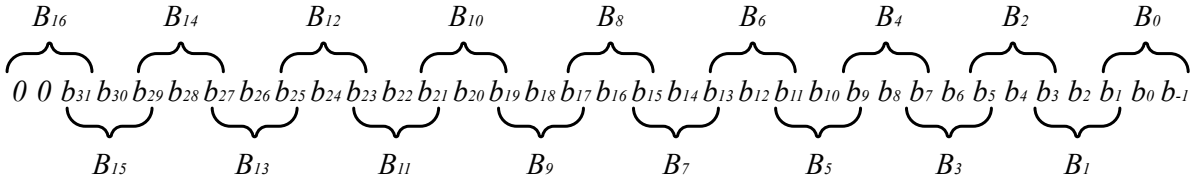

Figure 90: 32-Bit Multiplier Booth Radix-4 Encoding

the selected partial products.

In our CPU design discussed in the next chapter, three control signals are connected to this block: Shift, No-Shift and Negative. The Negative signal inverts the multiplicand thus deriving a one's complement value to which a '1' will be added later in the tree to convert this to two's complement. The No-Shift and Shift control signals cause the multiplexer to select either the original or a 1-bit shifted version of the multiplicand.

Partial Product Selection Table		
Multiplier Bits	Selection	Multiplicand Output
000	+0	0
001	+Multiplicand	No Change
010	+ Multiplicand	No Change
011	+ 2 x Multiplicand	<<1
100	- 2 x Multiplicand	<<1, 2's Complement
101	- Multiplicand	2's Complement
110	- Multiplicand	2's Complement
111	-0	0

Table 14: Partial Product Selection Table

Figure 91 is a typical clocked Boolean modified-Booth partial product generator [124]. The circuit is divided into a Booth decoder part and a Booth selector part.

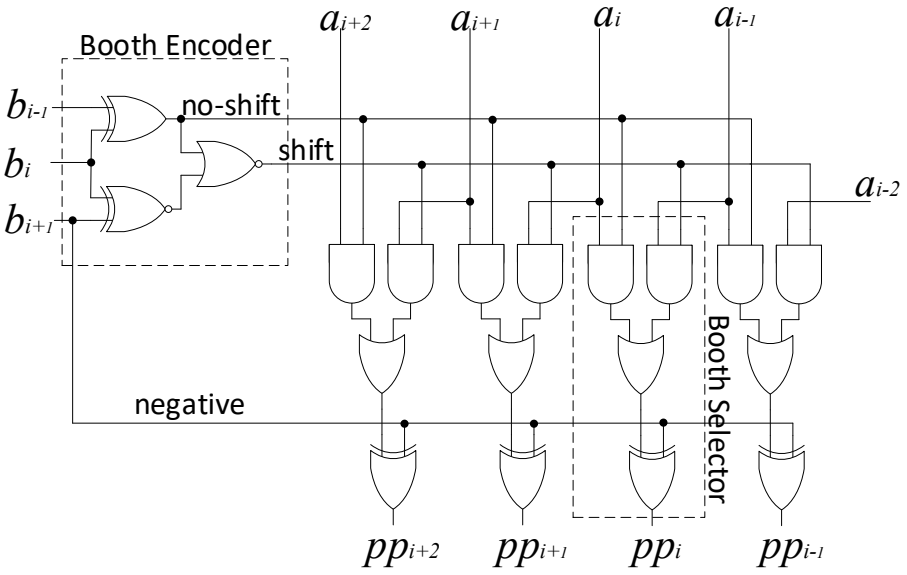

Figure 91: Radix-4 Booth Decoder/Selector

For the NCL modified-Booth partial-product generation, Smith suggested in [19] (Chapter 3) that the optimized equations can be implemented using TH54w32 and TH54w22 cells. However, that optimization is the just one example for the MSB bit of the partial-product generation and the design has only one input multiplicand (MD) bit and three selection bits (MR2, MR1, MR0). As explained in Figure 91, the radix-4 Booth encoding Partial-Product generation has three selection inputs to the Booth encoder from the multiplier and two input bits for the

Booth selector: the non-shifted and 1-Bit shifted bits of the multiplicand. The examples shown in [19] simply illustrate to show how to use Karnaugh maps to optimize the dual-rail NCL circuit. Other Booth multiplier designs have been published from the same group [125] [126] along with a 4x4 simple unsigned Booth unit, which is the same as a radix-4 modified Booth. That work has proposed and compared three solutions termed *'straightforward'*, *'dual-rail recoded'* and *'quad-rail recoded'*. Subsequently in this thesis, instead of the previously proposed dual-rail or quad-rail signaling schemes, a group of single-rail control signals (called 'One-Hot' control signals) have been proposed and used to select the output products on the Booth encoder stage.

The dual-rail NCL Booth Encoder (Figure 92) has two additional outputs compared to the Boolean version of Figure 91. The additional *zero* and *positive* outputs are necessary because, in this single rail one-hot encoding scheme, at least one output must be active at all times (as opposed to the Boolean case where the default is all signals inactive). The signals required to control the multiplicand outputs were listed previously in Table 14.

Figure 93 shows the fundamental dual-rail NCL Booth selector using NCL based multiplexer (two TH22 and one TH12 gates) and Figure 94 shows its optimized circuit. Although the logic depth of the NCL Booth selector is the same as the clocked Boolean implementation, because it is dual-rail, its gate count is double.

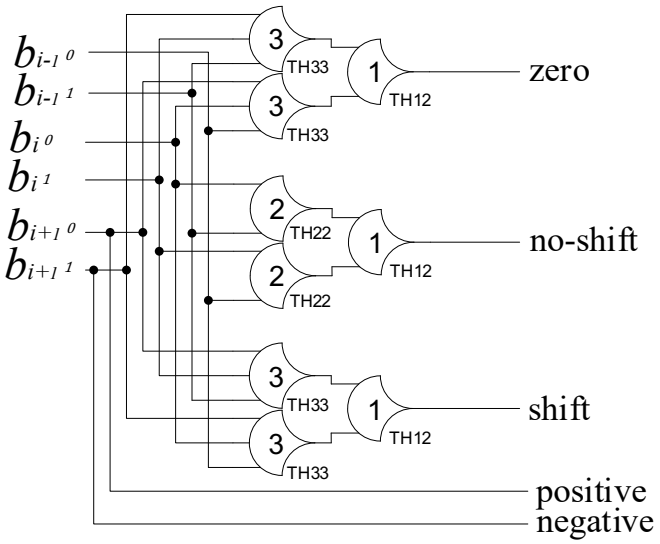

Figure 92: NCL Booth Encoder

Figure 95 shows the overall block diagram of the Booth multiplier. The detailed circuit design of Booth encoder and selector blocks have been discussed above so the next block to

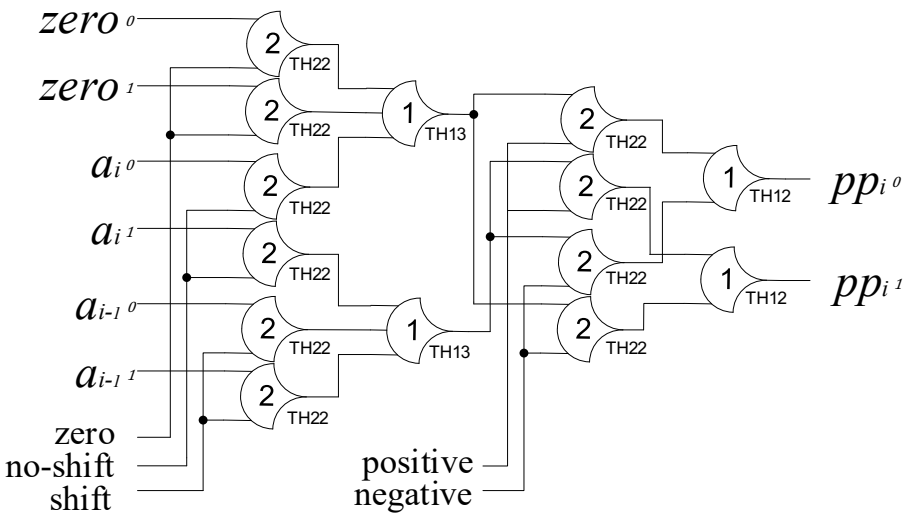

Figure 93: NCL Booth Selector using TH22 and TH1n NCL Multiplexer

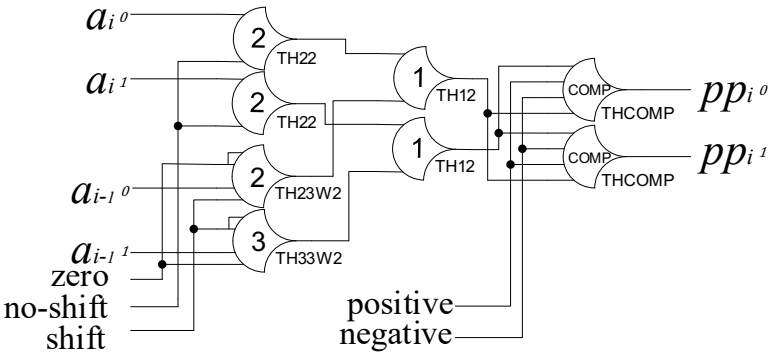

Figure 94: Optimized NCL Booth Selector

consider is the compressor. In many of these cases, the partial products are compressed using a tree architecture such as Wallace [122] or Dada [127] tree. It is typical to create these trees using compressor circuits, like 3:2 or 4:2 compressors. The 3:2 compressor circuit is identical to the full adder while the 4:2 compressor is functionally the same as two cascaded full adders. In a typical clocked Boolean design, dedicated 4:2 compressor cells are supported by the foundry as part of their standard cell library. The objective is to reduce the size and increase the compressor performance. However, in NCL there are no dedicated 4:2 compressor cells. Only 3:2 compressors (i.e., full adder blocks) were used for the tree as it was more efficient than creating 4:2 compressors with their two cascaded full adders. For the CPU multiplier in this work, a Wallace-Tree architecture was used as shown in Figure 96. This has 17 partial products and a six stage 3:2 Compressor reducing the partial products to the final sum and carry.

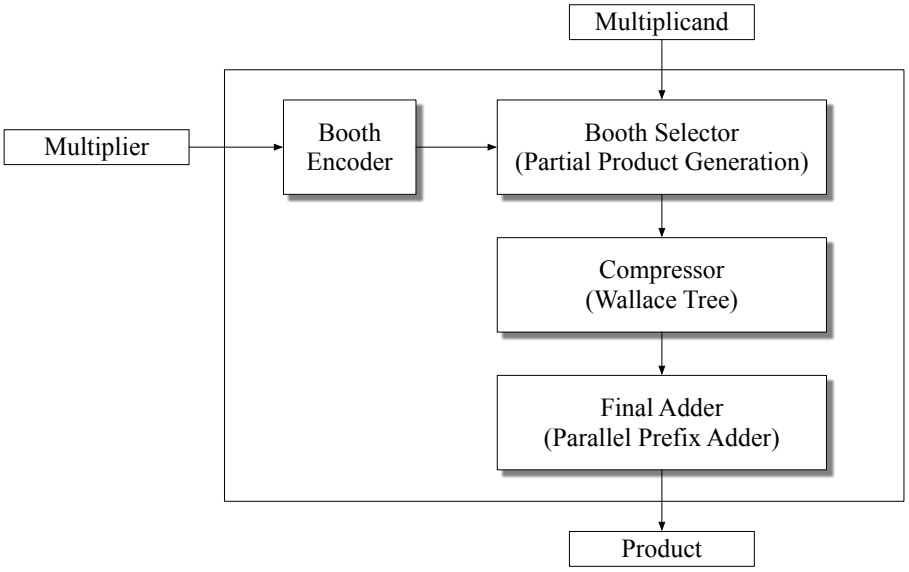

Figure 95: Booth Multiplier Block Diagram

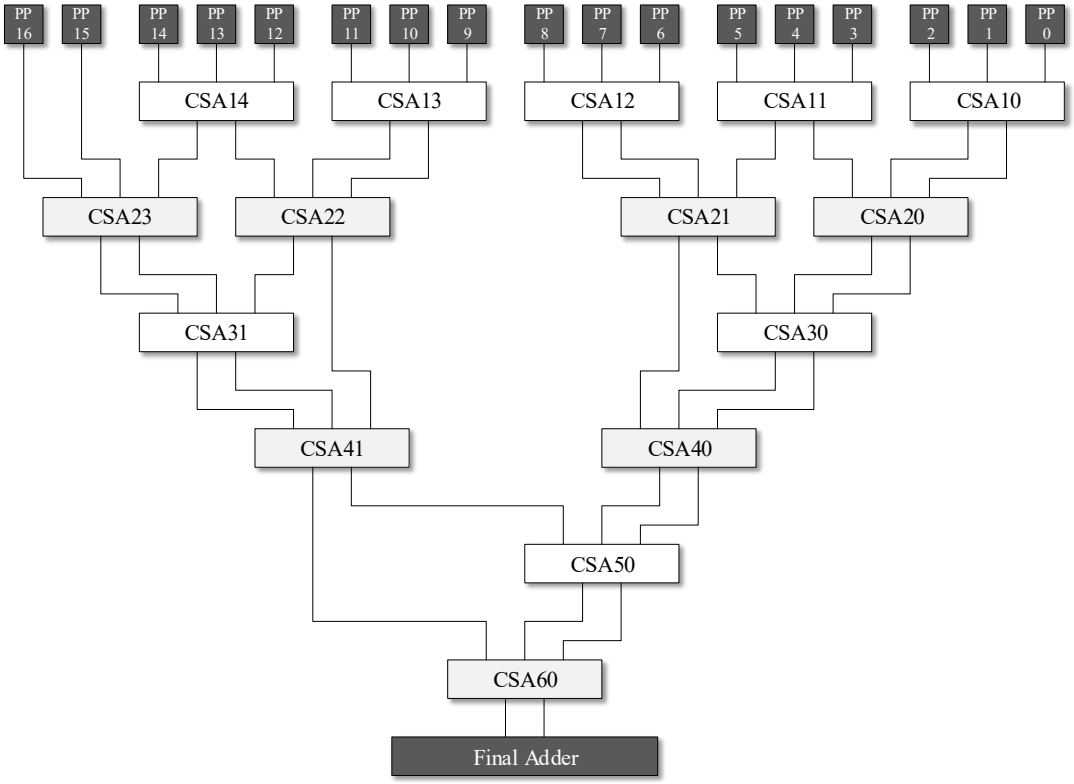

Figure 96: 32bitx32bit Modified Booth Multiplier Wallace Tree

Figure 97 illustrates the details of the 32-Bit radix-4 modified-Booth multiplier Wallace-Tree bit map table. "PP" represents Partial Product, "SM" is Sign-MSB and "SL" is Sign-LSB. The SM and SL values are decided by the operation (Signed/Unsigned and ADD/SUB) and input multiplier and multiplicand sign values. Because the RISC-V CPU described in the next

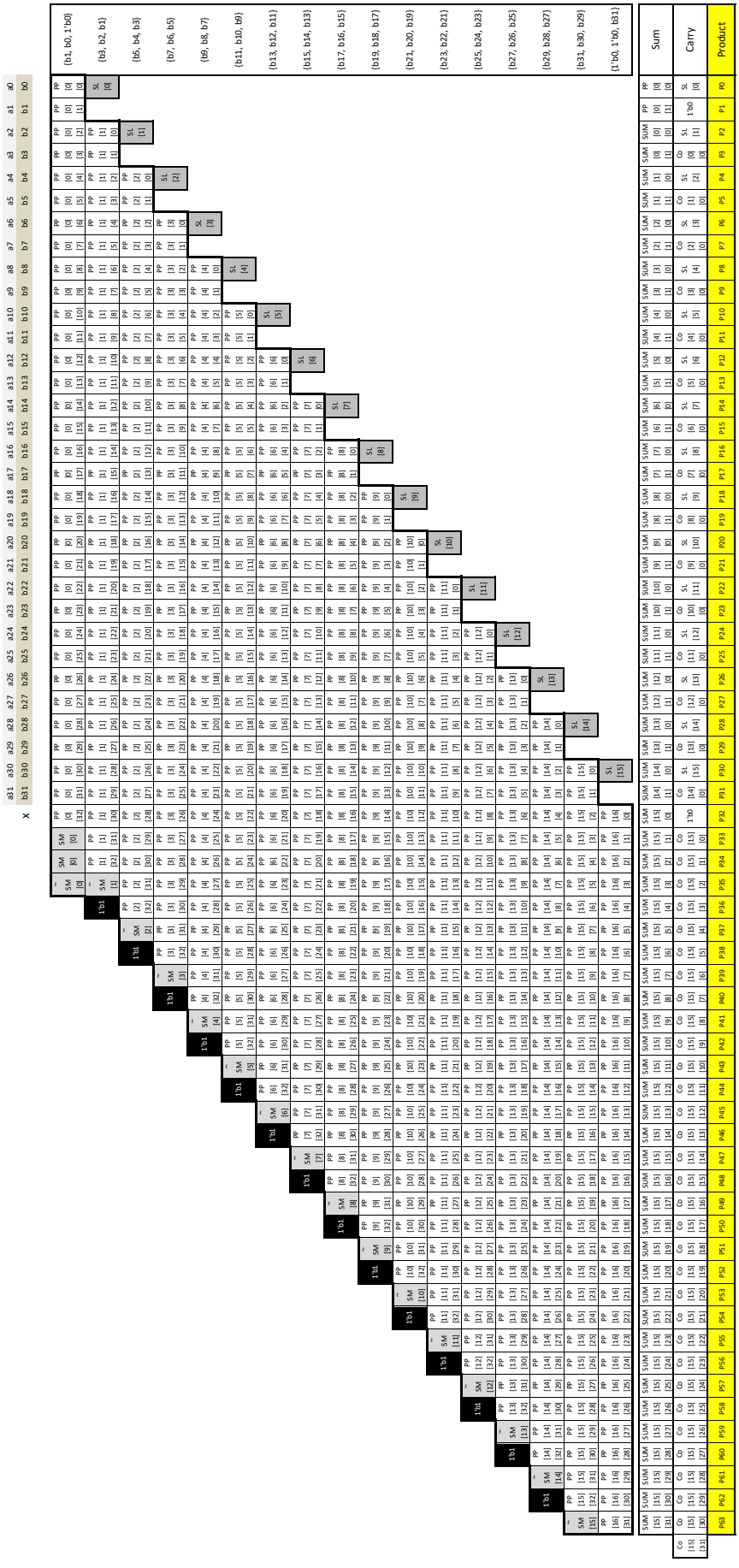

Figure 97: 32-Bit Radix-4 Booth Multiplier Wallace Tree BitMap Table

chapter requires signed multiplication, the table also includes the signed data processing using two's complement format.

The last stage, which comes after after the compressor tree, is the Final Adder, which adds together all of the partial sum and carry components (i.e., Carry-Save format) to generate a binary product output. As discussed in the previous section, multiple 64-Bit adder architectures were designed and tested as the Final Adder for this application.

NCL Modified Baugh-Wooley Multiplier

Baugh-Wooley algorithm was introduced by Charles R. Baugh and Bruce A. Wooley in 1973 [119]. The algorithm was developed for high-speed, two's complement, m-bit by n-bit parallel array multiplication. It maximizes the regularity of the multiplier and allows all the partial products to have positive sign bits. This means that, instead of subtracting the partial products that have negative signs the negation of the partial products can be added so that the architecture uses only addition and AND functions. This multiplier generates all the possible partial products first using AND terms, which are then sent through the adder array with the carry chain to the next most significant bit at each level. Because the conventional Baugh-Wooley still has an array architecture, it is still slow and has a larger area compared to other tree based multiplier architectures. This work employed a tree compressor structure with Baugh-Wooley partial product generation called the Modified Baugh-Wooley multiplier. This Modified Baugh-Wooley has no partial product reduction scheme and the compressor tree is therefore almost twice as large as the radix-4 modified Booth multiplier.

To replace subtraction in the signed multiplication algorithm, Baugh and Wooley added two final rows for the sign bits (MSB) of X and Y, and suggested adding the negation of the partial products for negative signs.

This is the equation of negation for subtraction:

$$2^{n-1}\left(-0 \bullet 2^{m} + 0 \bullet 2^{m-1} + \sum_{i=0}^{m-2} x_{n-1} y_i 2^i\right) \tag{4.10}$$

And this is replaced with the addition suggested by Baugh and Wooley [119]:

$$2^{n-1}\left(-2^{m} + 2^{m-1} + \bar{x}_{n-1} 2^{m-1} + x_{n-1} + \sum_{i=0}^{m-2} x_{n-1} \bar{y}_i 2^i\right) \tag{4.11}$$

The actual design is quite straightforward and emerges directly from the equations. The implementation uses dual-rail AND and NAND gates for the partial product generation. Figure 98 shows the NELL code for the 32-Bit NCL modified Baugh-Wooley multiplier partial product generation. In NCL, the NAND function is identical to AND with its dual-rail output signals swapped such that rail[0] becomes rail[1] and vice versa. In this NELL code, the module **NellAND** is equivalent to the Figure 83 dual-rail NCL AND.

```
////////////////////////////////////////////////////////////
// Partial Product Generation X AND Y
////////////////////////////////////////////////////////////
dual pp [WIDTH-1:0][WIDTH-1:0];

for i=0:WIDTH-2{
   for j=0:WIDTH-2{
      NellAND #n0(1)(x[j], y[i], pp[j][i]);
   }
}

dual pp_last_vertical[WIDTH-2:0];
for i=0:WIDTH-2{
   NellAND #n0(1)(x[i], y[WIDTH-1], pp_last_vertical[i]);
   pp[i][WIDTH-1] = {pp_last_vertical[i]/1, pp_last_vertical[i]/0};
} // For the last vertical line (Y_max, NAND)

NellAND #n0(1)(x[WIDTH-1], y[WIDTH-1], pp[WIDTH-1][WIDTH-1]);
  // For the X_max x Y_max (AND)

dual pp_last_horizontal[WIDTH-2:0];
for i=0:WIDTH-2{
   NellAND #n0(1)(x[WIDTH-1], y[i], pp_last_horizontal[i]);
   pp[WIDTH-1][i] = {pp_last_horizontal[i]/1, pp_last_horizontal[i]/0};
} // for the last horizontal line (X_max, NAND)
```

Figure 98: NELL Code for NCL Baugh-Wooley Multiplier
Partial Product Generation

Like other high-performance multiplier architectures, this NCL based 32-bit modified Baugh-Wooley multiplier has a partial product generation part (PPG), a Wallace-Tree carry save adder part (the compressor) and a 64-bit final adder (using ripple carry adder or other parallel prefix adders). Figure 99 shows the 32bitx32bit modified Baugh-Wooley multiplier Wallace tree architecture. Again, only 3:2 compressors are used in this tree as was done for the modified Booth multiplier. It employs eight stages of 3:2 compressors whereas the Booth style has six stages. The tree is almost twice the size of the modified Booth because the partial product count (32) is almost twice that of the Booth (17).

The modified Baugh-Wooley multiplier uses the same 64-Bit adder as the Booth case in its final stage. Three types of final adder were tested: Ripple-Carry Adder, Kogge-Stone Adder and Han-Carlson Adder.

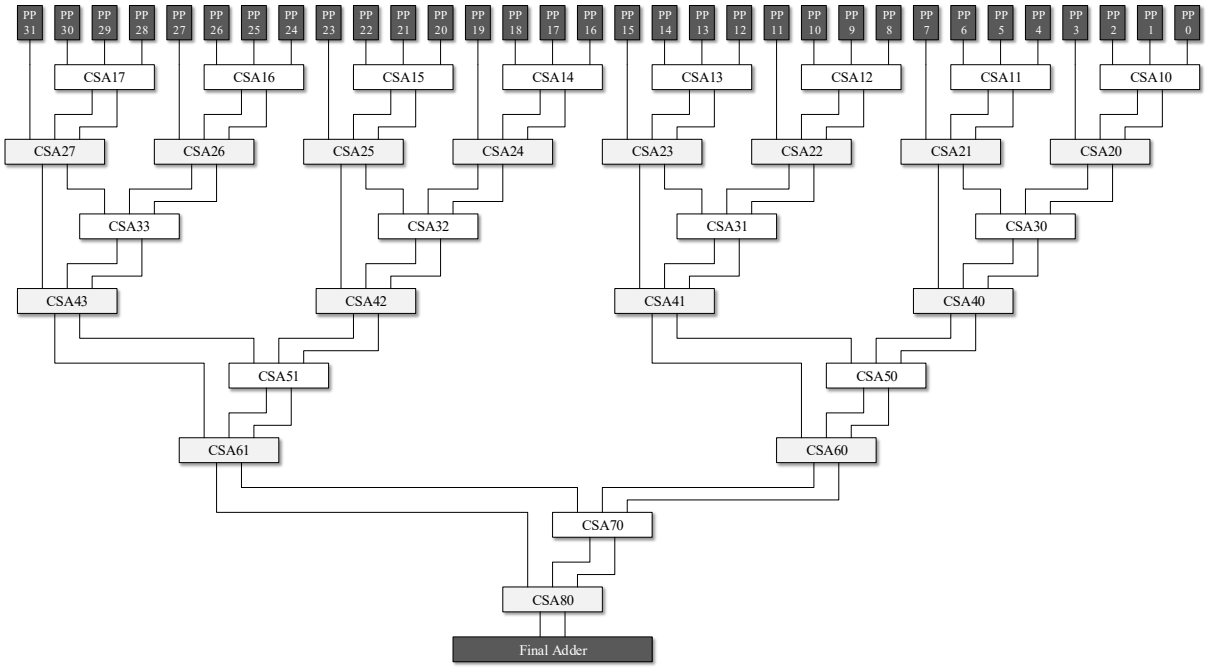

Figure 99: 32bitx32bit Modified Baugh-Wooley Multiplier Wallace Tree

4.2.3 Multiplier Throughput and Area Comparison Results

In this section, we analyse and compare the performance, power and area (PPA) results between all of the multipliers designed for this work. Table 15 shows the comparison results for the multipliers in terms of throughput and area for each architecture. As already mentioned, these simulations were performed using 28nm UTBB-FDSOI device models with a supply set to 1V. The circuits were designed using NELL and imported to Cadence Virtuoso and simulated using Cadence Ultrasim and NC-Verilog. By applying the same 100 input vectors in each case, the average throughput over each input to output path was measured. The first two rows (the Modified Booth and Modified Baugh-Wooley) were designed without a Wallace-Tree network but with array-based partial product addition. The Vedic multiplier exhibited better performance than the Modified Baugh-Wooley but worse than the Modified Booth. Further, Vedic has a larger area compared to the Modified Booth and Modified Baugh-Wooley, indicating that it does not really offer any significant advantages over conventional techniques such as Modified Booth.

The Modified Booth multipliers with Wallace-Tree were set up with options for 1-stage, 3-stage or 4-stage pipelining. The performance was seen to increase in proportion to the pipeline depth but the gate count also increased because of the larger number of register stages. The Modified Baugh-Wooley was tested with two options: with a Kogge-Stone final adder and

with a simple Ripple-Carry final adder. The ripple carry adder exhibits better performance and smaller area and its superior average case performance clearly illustrates the benefits of this simpler approach for this 64-Bit final adder design. The Modified Booth multiplier has slightly better performance and a smaller cell count than Modified Baugh-Wooley and, as a result of these experiments, it was decided to use the Modified Booth approach for the NCL CPU design described in the next chapter.

Table 15: 32-Bit Multiplier Comparison Results

32-Bit NCL Multiplier	Throughput and Latency Comparison Results			Area Comparison Results
	Cycle-Time	Throughput	Pipeline Depth	NCL Gates
Modified Booth	5,538 ps	181 MHz	1	6,075
Modified Baugh-Wooley	9,280 ps	108 MHz	1	10,084
Vedic	6,156 ps	162 MHz	1	13,795
Modified Booth with Wallace-Tree (1-Stage Pipeline) with Ripple-Carry Adder	3,830 ps	261 MHz	1	6,276
Modified Booth with Wallace-Tree (3-Stage Pipeline) with Ripple-Carry Adder	2,582 ps	387 MHz	3	8,465
Modified Booth with Wallace-Tree (4-Stage Pipeline) with Ripple-Carry Adder	1,895 ps	528 MHz	4	9,416
Modified Baugh-Wooley with Wallace-Tree (1-Stage Pipeline) with Kogge-Stone Adder	4,374 ps	229 MHz	1	8,414
Modified Baugh-Wooley with Wallace-Tree (1-Stage Pipeline) with Ripple-Carry Adder	3,930 ps	254 MHz	1	6,856

4.2.4 Two-Dimensional Pipelined NCL Multipliers

As discussed in the previous Adder section, fine grained two-dimensional (2D) pipelining offers a throughput advantage to NCL. In this section, the 4-bit unsigned array multiplier is used as a case-study to discuss the advantages of 2D pipelining in the dual-rail NCL multiplier and the trade-off between performance, power and area. Finally, a 64-Bit Two-Dimensional Modified Baugh-Wooley Multiplier is designed and analysed using various 2D pipelined final adder designs.

Two-Dimensional Array Multiplier[3]

[3]This work has been published as "Area performance tradeoffs in NCL multipliers using two-dimensional pipelining" [128].

Component Analysis for 2D Pipelining

The adder blocks (Full-Adder and Half-Adder with Output Registers) can be optimized by integrating its register functions with the combinational gates. Figure 100 shows the dual-rail full adder with integrated registers (called the THACK gate) that generates the output Acknowledge signal. In NCL, the term Acknowledge is generally used for the inverted output of the completion signal, therefore this THACK gate is the inverted output of the THCOMP cell. All inputs are not created equal in NCL gates like the THCOMP etc., which contrasts with standard Boolean logic gates where the inputs are interchangeable. In a similar manner, the optimized half adder with integrated registers and completion logic is shown in Figure 101. A potential disadvantage here is the need for a five transistor PMOS stack in the TH45w2 gate that drive the Sum outputs. These gates are not part of the fundamental NCL set, but could be implemented as a custom function if necessary. Five level p-type stacks of this sort are generally avoided in CMOS standard cell design as they tend to result in slow rise-times and longer propagation delays. However, in our experiments using the 28nm UTBB-FDSOI process, the delay of the adder with integrated registers proved to be virtually identical to its non-integrated counterpart, even before any attempts were made to optimize the transistor sizes. The dual-rail AND input complete module (Figure 102) with integrated registers is used to generate the partial product at the input of the multiplier.

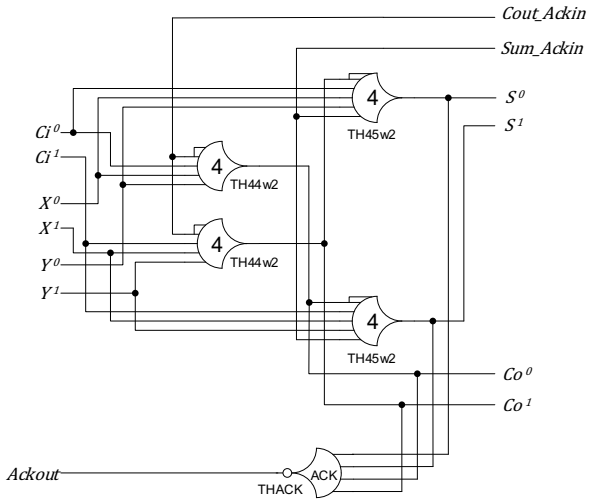

Figure 100: Optimized Full-Adder with Integrated Registers

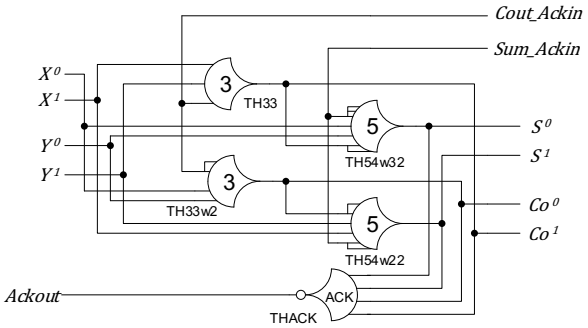

Figure 101: Optimized Half-Adder with Integrated Registers

The dual-rail register module shown in Figure 103 is used to control the basic data flow timing logic through the 2D pipelined multiplier. Completion detection in this case requires only the small TH12B gate (identical to a four transistor Boolean NOR2). As a result,

this fine-grained completion logic uses many fewer transistors than the full completion circuit required by the non- and 1D pipelined system.

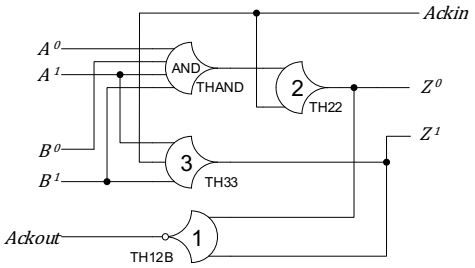

Figure 102: Dual-Rail AND with Integrated Registers

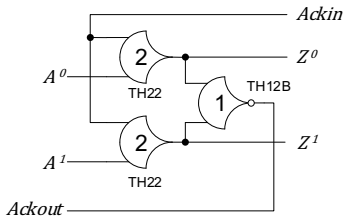

Figure 103: Dual-Rail Register

Non-pipelined architecture

The Non-Pipelined Array Multiplier Architecture (Figure 104) has no internal register components making the multiplier entirely combinational. As such, it represents the slowest NCL technique but occupies the least area. As already mentioned, in common with other DI design styles, NCL exhibits average-case performance so the processing time of the multiplier will be entirely input data dependant. The primary constraint here is the number of propagating carry values within the ripple-carry adder blocks. In cases where a constant cycle time is required (e.g., in response to a fixed external sampling rate) then this logic needs additional registration.

The block diagram of the 4x4 dual-rail multiplier in Figure 104 shows the data flow in blue and handshaking signals in red. The multiplier components shown in yellow are dual-rail AND modules that are input-complete. In this case, these are separate from the data registers. This multiplier block uses just two 8-bit dual-rail registers at its input and output ports.

1D Pipelining with Ripple-Carry chains

Figure 105 shows the detailed diagram of a 1D Pipelined architecture. Each pipeline stage includes a ripple-carry adder and their carry chains. As for the non-pipelined architecture above, this architecture also exhibits average-case delay behaviour. The multiplier has five 1D pipeline stages with each stage exhibiting a different word size. The input and output registers are the same size as the non-pipelined case while the remaining three intermediate registers are 16, 14 and 11 dual-rail lines respectively. The completion detection circuit here is relatively large, requiring connection to all 8 dual-rail signals (i.e., 16 bits). The delay time of this completion path (called its *reverse latency*) is the main drawback of this architecture [129]. It can be seen that its structure is similar to the 12x12 array in [130], which exhibited a 76.5%

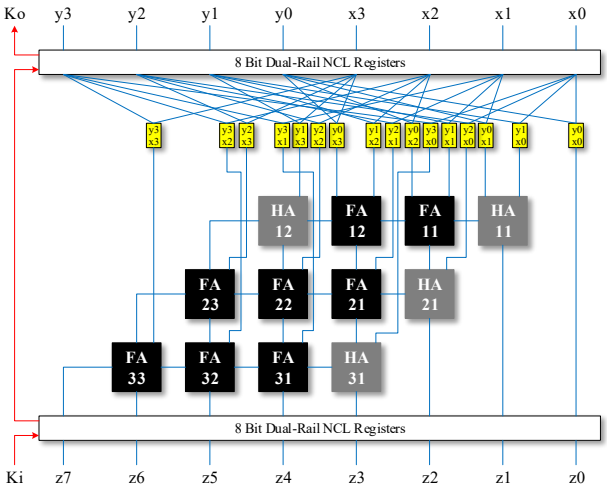

Figure 104: Dual Rail Array Multiplier Non Pipelined

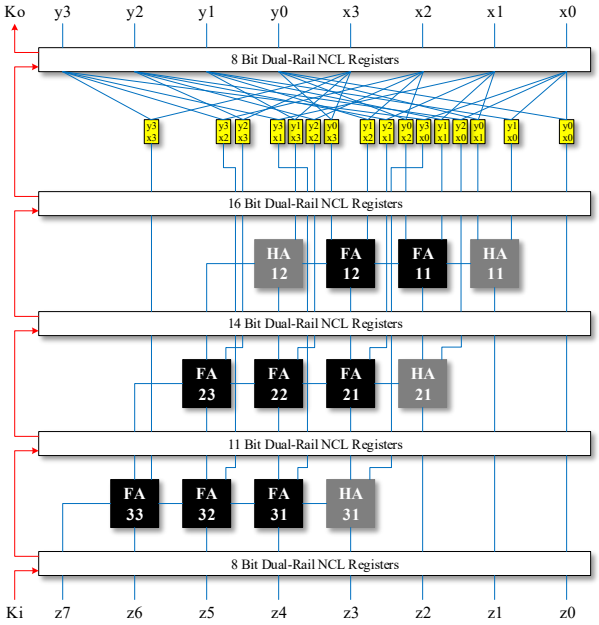

Figure 105: 1D Piplelining with Ripple Carry

throughput improvement compared to the equivalent non-pipeline case. However, [130] did not consider the timing improvements available as a result of the average case delay behaviour of asynchronous techniques such as NCL.

1D Pipelining without Ripple-Carry chains

Figure 106 shows a 1D pipelined multiplier with no combinational ripple-carry chains. All of the adder modules in this architecture are separated by dual-rail registers that include the carry-chains. In this case the multiplier comprises ten stages, each with a different register size. As in the previous 1D case (Figure 105), the completion detection circuit is large, also comprising a module monitoring all eight output pairs. This completion detection logic delay is in the critical path and therefore governs the total performance. The architecture has more latency than Figure 105 but better throughput. It is also worth noting that it exhibits a fixed delay regardless of the input data.

2D Pipelining with Triangle Buffers

Figure 107 shows an example of a 2D pipelined multiplier with input and output Triangle Buffers, which are formed from sequences of dummy registers with no combinational logic. Compared to 1D pipelined architectures, this organization can greatly reduce the delay in the completion paths. The critical path in this architecture is the sum path through the full-adder blocks as these optimized NCL dual-rail adder circuits have two gate delays for the sum output

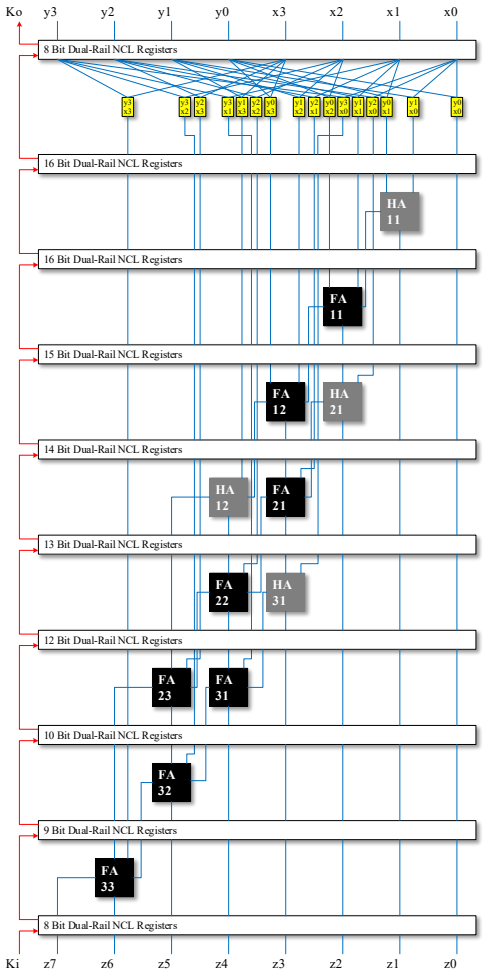

Figure 106: 1D PIpelining without Ripple Carry Chains

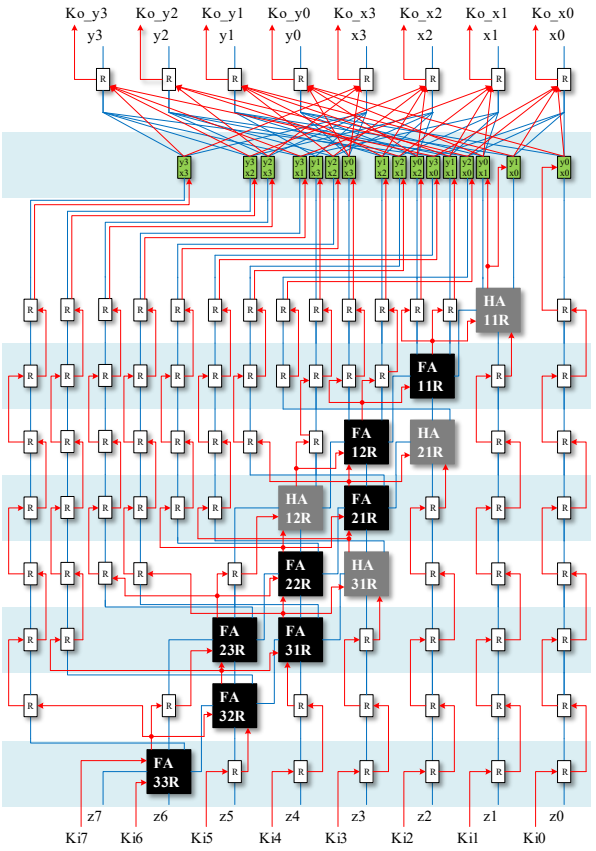

Figure 107: 2D Pipelining with Triangle Buffers

and a single gate delay for the carry. In Figure 107, the green components are dual-rail AND input complete modules that include dual-rail registers and their completion logic. The components labelled 'R' comprise dual-rail pipeline registers. The half and full adder components also include dual-rail registers and their completion logic.

One advantage here is that the delay time of this multiplier is very predictable while the non-pipelined and 1D pipelined case depend on the number of active ripple-carry bits so exhibit the occasional long delay. The 2D pipelining architecture has balanced latency across all its data paths so it can be easily connected to other non-2D pipelined modules. However, to connect correctly to 1D or non-pipelined modules, may require additional input and output alignment register blocks.

2D Pipelining without Output Triangle Buffers

Figure 108 shows a 2D pipelined organization where the output buffers have been removed.

As a result, while the input and output latencies are not balanced, this reduced circuit has improved area, speed and power. While the circuit can be used seamlessly with other 2D pipeline architectures, care has to be exercised when connecting these structures to non-pipelined or 1D pipelined stages as their latency behavior is quite different. Additional stage delays or more complex completion logic would be necessary to ensure that all data bits in a result exhibit the same latency between the different architectures.

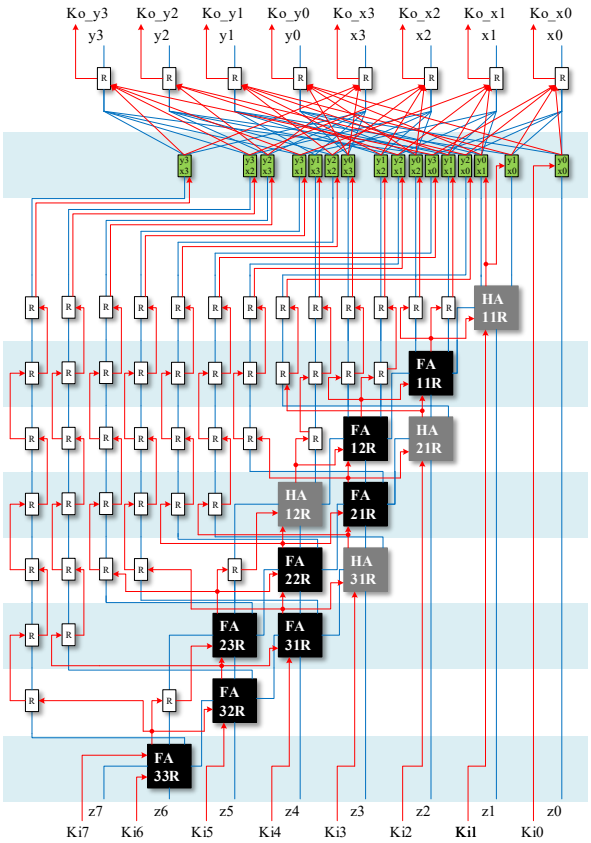

Figure 108: 2D Pipelining without Output Triangle Buffers

Throughput, Area and Power Comparison Results

As mentioned above, the comparative results for throughput and area for each multiplier architecture were derived from simulations performed using 28nm UTBB-FDSOI device models with a supply set to 1V. The designs were implemented using Structural Verilog HDL and imported to the Cadence Virtuoso and simulated using Cadence Ultrasim and NC-Verilog.

Table 16 shows comparative cycle-time, throughput and latency results for various non-pipelined and pipelined architectures. It is clear that the 2D pipelining case supports a performance increase of more than twice compared to the non-pipelined case. It has to be noted that these results are derived using a small (4x4) multiplier circuit. However, at larger array

sizes (e.g., 16x16 or 32x32), the relative performance gap between each architecture increases. It can be seen that the cycle-time of the 2D pipelined case is independent of size. In this case, only the number of pipeline stages increases. It can also be seen that the 2D pipelined architecture uses more than three times the transistor count of the non-pipelined circuit (Table 16). This indicates that 1D pipelining with ripple-carry would be good choice if the balance between speed and area was biased in favour of speed.

The 2D pipelined architecture uses almost three times the power of the non-pipelined architecture (Table 16). Power was measured with a 500MHz (2nS) input transition rate applied under the same conditions to each architecture. It is also of note that this UTBB-SOI process results in extremely small values of leakage power.

Table 16: 4x4 Array Multiplier Results Comparison

4x4 Array Multiplier	Throughput and Latency Comparison Results			Area Comparison Results		Power Consumption Comparison Results		
	Cycle-Time	Throughput	Pipeline Depth	Gates	Transistors	Total Power	Dynamic Power	Static Power
Non-pipelining	1648.55 ps	606 MHz	2	136	1620	77.4 uW	77.3 uW	58.5 nW
1D Pipelining with Ripple-Carry Chains	1032.50 ps	968 MHz	5	276	2748	138.5 uW	138.4 uW	100.0 nW
1D Pipelining without Ripple-Carry Chains	896.62 ps	1.115 GHz	10	494	4710	233.6 uW	233.4 uW	164.9 nW
2D Pipelining with Tri-angle Buffers	637.18 ps	1.569 GHz	10	439	5064	259.9 uW	259.6 uW	229.0 nW
2D Pipelining without Output Tri-angle Buffers	637.18 ps	1.569 GHz	10	361	4336	187.5 uW	187.4 uW	134.0 nW

Two-Dimensional Modified Baugh-Wooley Multiplier

For a conventional clocked design, the modified Booth multiplier has better performance and uses less area because of the compressed partial products using Booth encoding. However, in an NCL implementation, the size of the Baugh-Wooley multiplier was almost the same as the Modified Booth because of the larger partial product generation part [131]. Further, the pipeline depth just increased by one or two gate delays. The modified Booth multiplier also has a disadvantage in 2D pipelined designs because the partial product component has a large delay compared to the other circuits and it is difficult to partition into additional pipeline stages. Therefore we selected Baugh-Wooley for this complex 2D application (i.e., 32-Bit signed modified Baugh-Wooley multiplier with Wallace-Tree carry save adders). We used a Wallace tree organization because it has fewer pipeline stages compared to the normal array multiplier using Ripple-Carry adders and it is also faster. In the case of multipliers, it is also possible to merge the NCL registers with the combinational logic and these results in circuits of the form

of Figure 100 and Figure 101. The same Wallace-Tree architecture (Figure 99) was used for the 2D 32x32 Baugh-Wooley multiplier. This has 32 Partial Product modules and 30 3:2 compressors, which is equivalent to 15 4:2 compressors (CSA blocks). The Partial-Product generation part uses the same dual-rail NCL AND module with integrated registers (Figure 102).

Table 17 presents the comparison results of the 32x32 Baugh-Wooley multiplier with Wallace tree, where the Wallace tree is connected to the three versions of the final adder that were analysed here–the Ripple-Carry, Kogge-Stone, Han-Carlson (Table 12). It can be seen that the multiplier with non-pipelined RCA has higher performance compared to the parallel prefix adders because it exhibits average case carry propagation delays. However, the multiplier with Parallel Prefix Adders has better performance when they are used together with the 2D architecture. In the same way as the previous adder cases, the multiplier designed with HCA occupied less area compared to the KSA but both the area and performance results for the 2D architecture were similar.

Table 17: 32-Bit Baugh-Wooley Multiplier Comparison Results

32x32 Baugh-Wooley Multiplier with Wallace-Tree	Throughput and Latency Comparison Results			Area Comparison Results
	Cycle-Time	Throughput	Pipeline Depth	NCL Gates
Baugh-Wooley Multiplier with Ripple-Carry Adder(Non-Pipelining)	3,967 ps	252 MHz	1	6,856
Baugh-Wooley Multiplier with Kogge-Stone Adder(Non-Pipelining)	4,420 ps	226 MHz	1	8,414
Baugh-Wooley Multiplier with Han-Carlson Adder(Non-Pipelining)	4,622 ps	216 MHz	1	7,682
Baugh-Wooley Multiplier with Ripple-Carry Adder(1.5D Pipelining)	1,495 ps	669 MHz	11	15,770
Baugh-Wooley Multiplier with Kogge-Stone Adder(2D Pipelining)	1,175 ps	851 MHz	18	20,453
Baugh-Wooley Multiplier with Han-Carlson Adder(2D Pipelining)	1,162 ps	860 MHz	19	20,493

4.3 NCL Shifter Design

The shift function is a very important data-path element of any CPU Arithmetic Logic Unit and can also be used for Floating Point Unit implementation. A Barrel Shifter, for example, is typically implemented using only combinational logic (multiplexers) to perform its shift and rotation functions. The *Shift Amount* control determines how much the input data is shifted and

is implemented as the combination of 2^n-bits using 2:1 multiplexers. The depth and number of multiplexers required for the Barrel Shifter is calculated as follows (when n is the width of the shifter):

$$Depth = log_2 n$$
$$Number of Multiplexers = n \bullet \log_2 n \tag{4.12}$$

Therefore, the 32-Bit Barrel Shifter will be five stages deep and needs 160 2:1 multiplexers.

4.3.1 NCL Barrel Shifter

NCL shifter designs are divided into MUX-based and DEMUX-based styles [132]. Figures 109 and 110 present the dual-rail NCL 2:1 Multiplexer and dual-rail NCL 1:2 Demultiplexer circuits, respectively. In case of NCL, the shifter MUX and DEMUX circuits are similar and their gate counts are identical. Further, the cell driving capacitance are also almost the same, implying that the performance of each will be very similar. Therefore, for the shifter in the RISC-V CPU design, only the MUX-based design style was used.

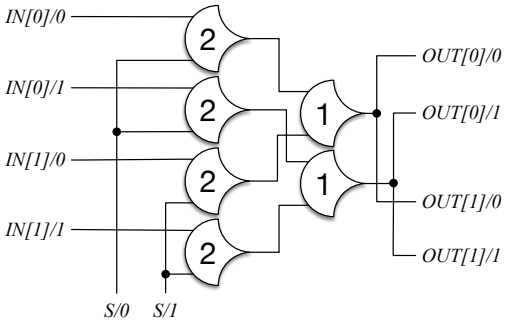

Figure 109: Dual-Rail NCL MUX

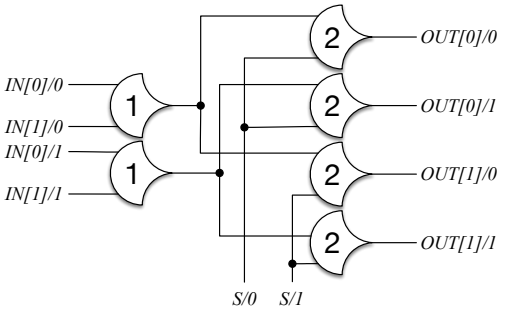

Figure 110: Dual-Rail NCL DEMUX

Figure 111 shows the 32-bit NCL MUX-based Barrel Shifter circuit that has 32-bit inputs and outputs and a 5-Bit shift control. The block uses 5 Multiplexer stages (shifting in turn by 1, 2, 4 , 8 and 16 bits). All ports and nets are actually dual-rail signals in this diagram.

4.4 NCL Register File Design

The register file is an array of processor registers and is one of the most important of the CPU blocks. RISC CPUs in particular make extensive use of their register file in that almost all operations are initiated by a register read and terminated after writing the result back to a register. The organization of the register file is strongly related to the ISA. The RISC-V RV32IM

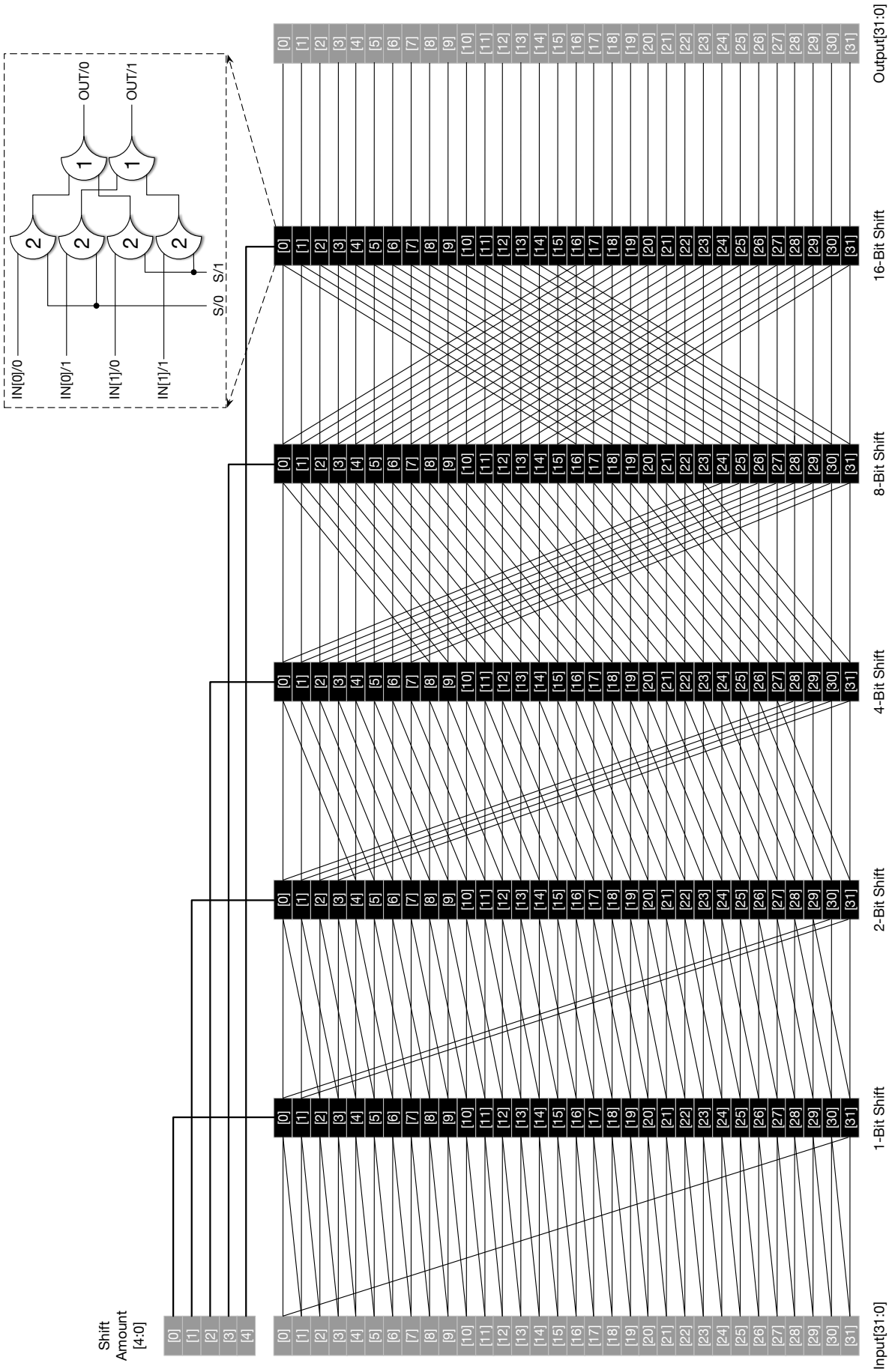

Figure 111: 32-Bit NCL MUX-based Barrel Shifter

instruction set used for this CPU design requires a 32-entry 32-bit register file. Register 0 is always zero (and is read only) and the ISA specifies one write port (destination register, RD) and two read ports (source registers, RS_1 and RS_2).

4.4.1 NCL Register File Organization

In clocked Boolean circuits, flip-flops or transparent latches are used for the default register file element. In contrast, dual-rail NCL requires a dual-rail register file design. The concepts behind dual-rail NCL register file structures were first introduced in [133] and [18].

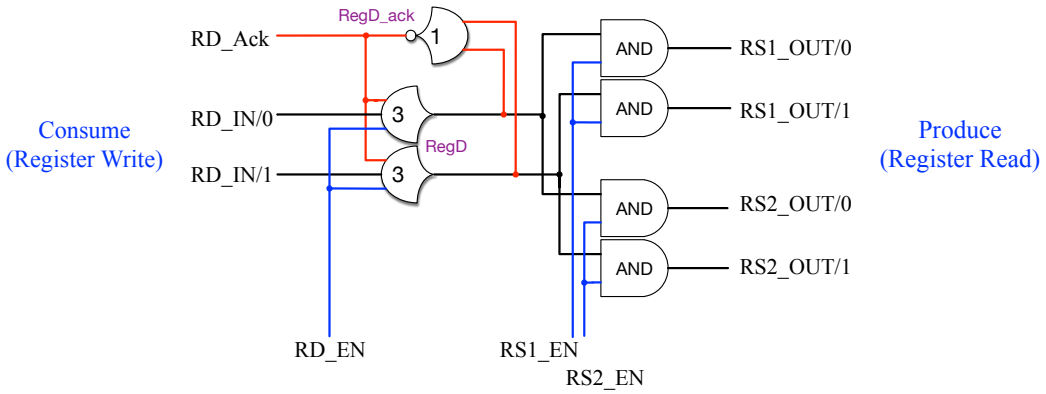

Figure 112: 1-Bit Dual-Rail NCL Register

Figure 112 shows a 1-bit slice of the NCL register file developed for this work. It comprises two basic parts labeled *Produce* and *Consume*. These use essentially the same concepts as the auto-produce structures used to generate constants into an NCL network, and auto-consume gates that are used to "pull" data from a (perhaps otherwise unused) gate output. The *Consume* region represents the file write port and corresponds to the destination RD of the ISA. This part interacts with the NCL handshaking (Request and Acknowledge) and, as for a clocked Boolean design, it terminates the instruction operation. The *Produce* part is the register read port, corresponding to the source (RS) of the ISA. Because we need two source register ports (RS_1 and RS_2), two dual-rail register read ports are required at the output of each register element. The Produce part initiates NCL handshaking and drives the output lines (RS1/2_OUT0/1) immediately when the enable signals (RS1_EN and RS2_EN) are asserted. No additional handshaking is required. Thus, this is the only point "officially" allowed to use a Boolean AND gate in the NCL implementation because the initiation timing can be started regardless of the NCL handshaking timing. This is equivalent to generating a constant into the network, with the exception that the constants in this case are the register contents. The AND

gates are directly connected to the output OR-tree of the register file read port. The registers must be set to NULL before writing DATA to activate the handshaking signals and to ensure that the registers take part in the NCL cycle of DATA and NULL. This register file structure has a large area disadvantage compared to the synchronous case as it uses seven NCL gates to perform the same function as a simple 1-Bit Flop or transparent latch with two output AND gates.

4.4.2 NCL Register File for RISC-V

In the NCL based RISC-V CPU design, we have four write ports from the execution modules and five read ports to the execution modules. The write ports are connected to the destination port (RD) through the dual-rail NCL multiplexer, and the read ports are connected from RS1 and RS2 ports through the dual-rail NCL demultiplexer.

Figure 113 shows the diagram of the dual-rail 32-entry 32-Bit NCL Register File. Again, in the RISC-V ISA, Register 0 (Left side Light Blue colour) is always zero and read only. Four input ports (left side) are connected from the "Load and Store Unit", "ALU", "Multiplier" and "Jumpreturn" execution modules. RD, RS1 and RS2 register selection inputs (left, bottom) are from the instruction decoder and the write control signals (RFWcontrol) select the proper input RD ports on their multiplexers. Output ports (right side) are connected to the "ALU", "Multiplier", "Branch Unit", "Load Store Unit" and "Program Counter" for their operands and the read control signals (RFRcontrol) select the proper output RS ports on their demultiplexers. The details of Register File module interfaces are explained in the next Chapter.

4.4.3 NCL Register File Write-Back Queue

[4] In NCL the execution time of each processor execution block, ALU, multiplier etc. will always be different. Even the input to output delay in the same execution block may differ substantially as the delay is input data dependent. Overall, these exhibit average-case delays rather than worst case, which can further complicate their design. This mandates the use of a suitable results forwarding circuit on the Write-Back side of the register file to guarantee its correct operation. The layout of these results forwarding circuit must include careful buffer sizing to maximize its performance.

Normally, in synchronous design, a FIFO (First In Fist Out) structure is used for the Write-Back Queue. However, in NCL the circuit itself behaves like a FIFO (called a *queue*) so

[4] This work has been published as "Design of asynchronous RISC CPU register-file Write-Back queue" [134]

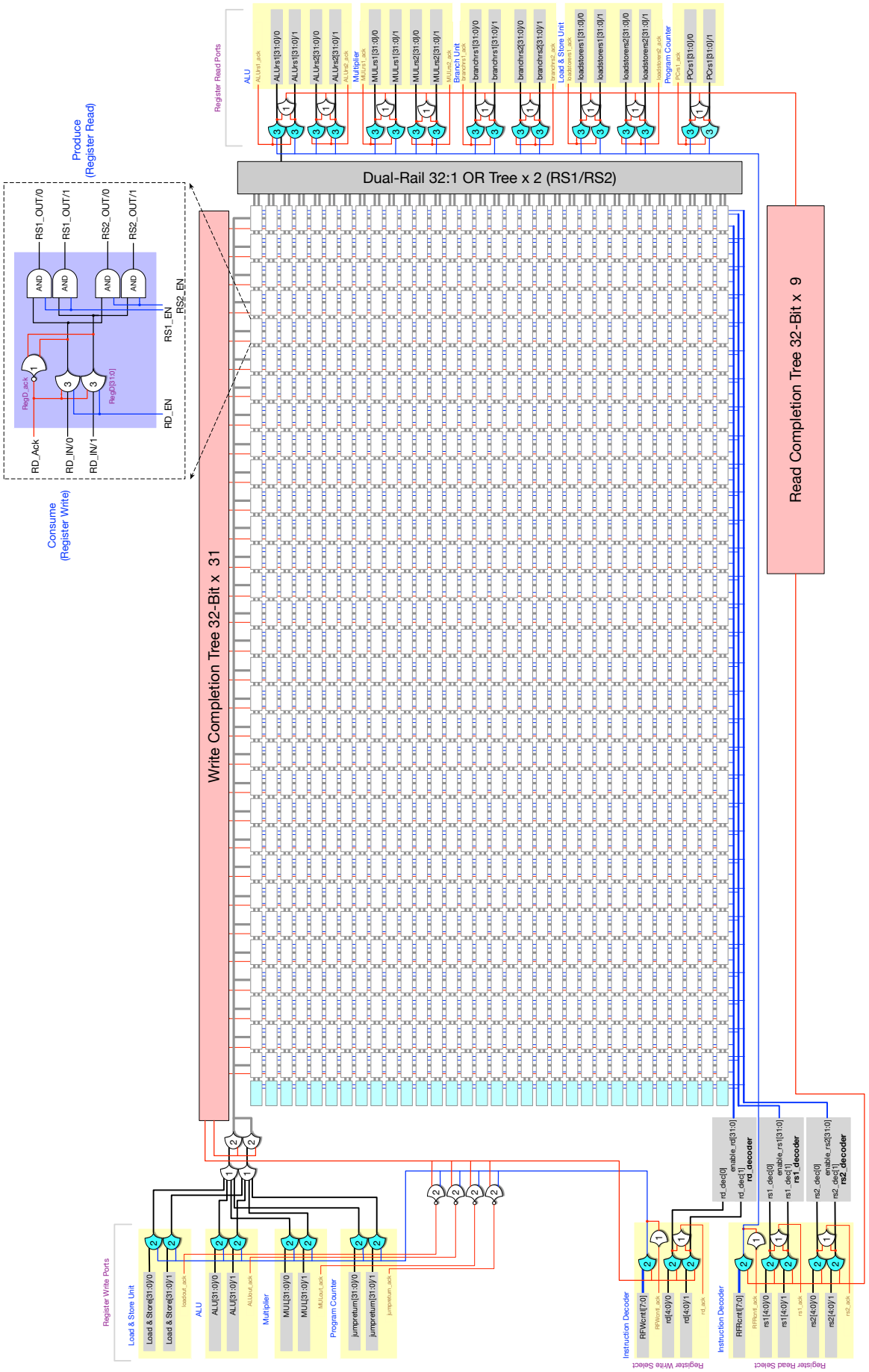

Figure 113: 32-Bit x 32Entry NCL Register File

real FIFO structures (i.e., with Read Counter and Write Counter) are not actually required for the Write-Back Queue. A shift register block is implemented using NCL gates and this operates in a way that is identical to a FIFO [134].

Figure 114 shows the simplified circuit diagram of an NCL based RISC CPU Write-Back Unit. The controls are derived from the instruction decoder and separate 32-bit dual-rail results emerge from each of the four separate execution units. The acknowledge handshaking signals are connected from the completion logic back to the execution units. An NCL fan-in steering circuit executes the multiplex function here. The block shown in yellow in Figure 114 is the Write-Back Queue.

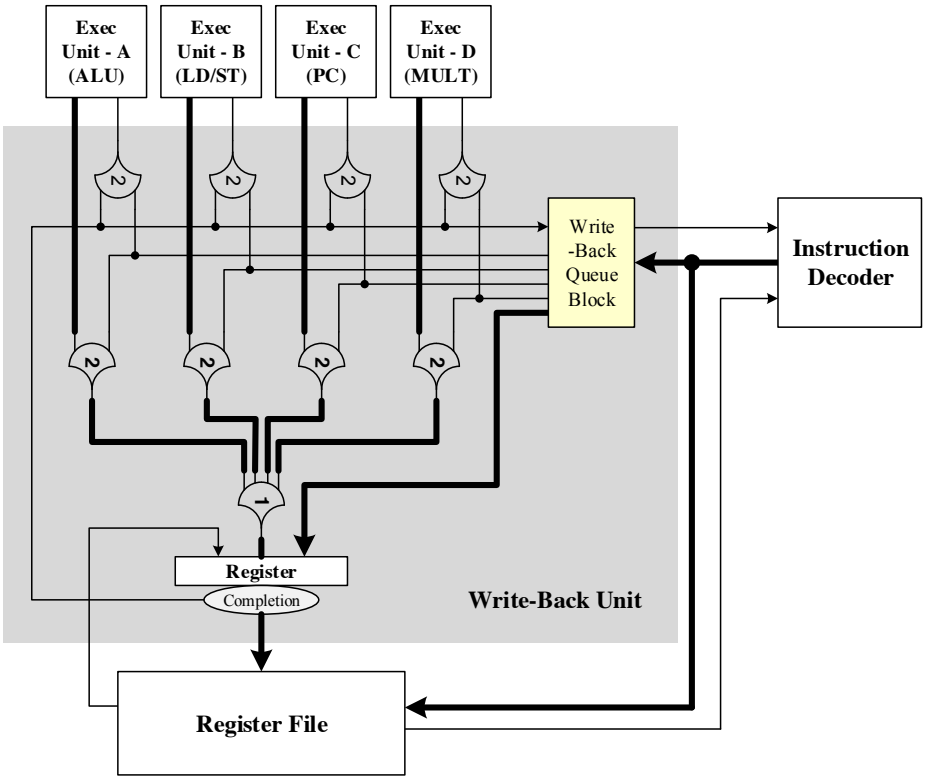

Figure 114: NCL Register File Write-Back Unit

Figure 115 illustrates the basic data flow of the NCL register chain. The bold lines represent "DATA (Value 1)" and the Light lines are "NULL (Value 0)". State (T1) is the reset state. All of the completion inverters have reset signals so that the data flow and the acknowledge flow status can be initialized to all NULL. In (T2) the reset is removed and all of the Acknowledge signals become DATA simultaneously. Note that the forward data path is still entirely NULL. At State (T3) an input DATA value is presented. As a result, this input value flows all

the way through the registers until it reaches the end of the Buffer. In the meantime, the DATA value is held active by the Ack Input line. At states (T4), (T5) and (T6), the input becomes respectively NULL, DATA and finally NULL again. By (T6) the *Buffer Full* status is asserted. The buffer undergoes a NULL-DATA-NULL-DATA cycle and the first input DATA will become present at the output when the Ack Input signal is asserted.

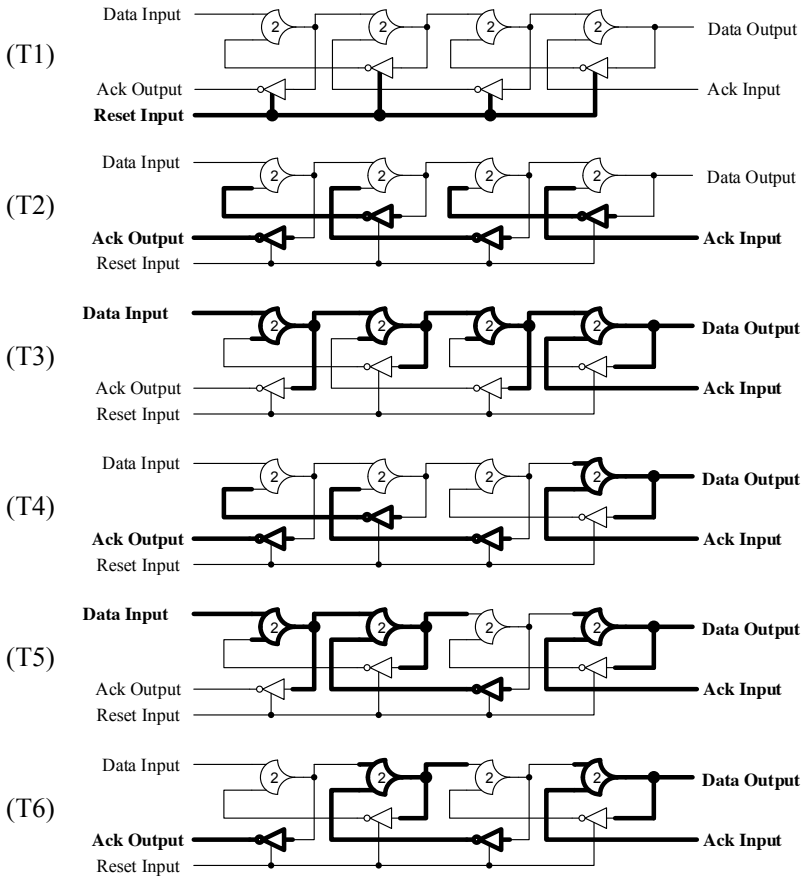

Figure 115: NCL Data Queue Register Data Flow

The execution unit delay time in the NCL based processor varies widely as its execution time depends on the input data. As a result, it is easy to lose the correct instruction order. A First-In/First-Out (FIFO) structure such as shown in Figure 114 is a good solution to maintain the correct instruction order. As also noted above, a register chain built using NCL naturally exhibits FIFO behaviour.

Figure 116 shows the circuit diagram of NCL data queue. This has 2-bit dual-rail width and 2 depth buffering functions. As in Figure 115, the circuit is a register chain and operates as a data queue, After Reset, if all the acknowledge signals are DATA, then input data

flows through to the end of the buffer. Therefore, the input DATA and NULL cycles are stacked in the Buffer waiting until the Read_Ack_In signal is becomes active.

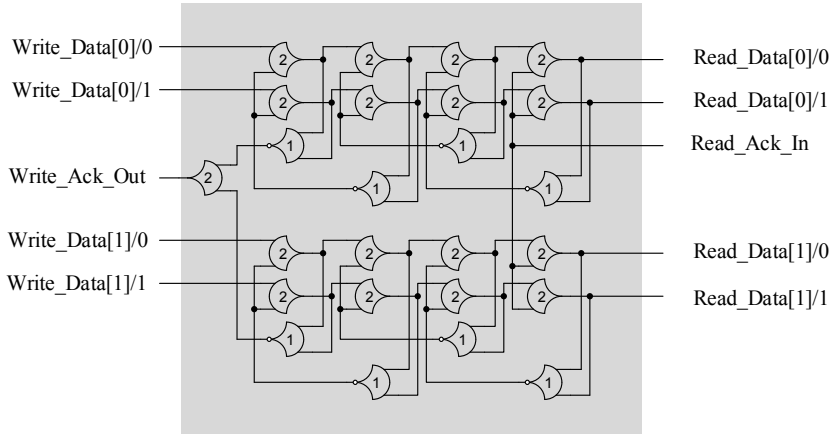

Figure 116: 2x2 NCL Data Queue

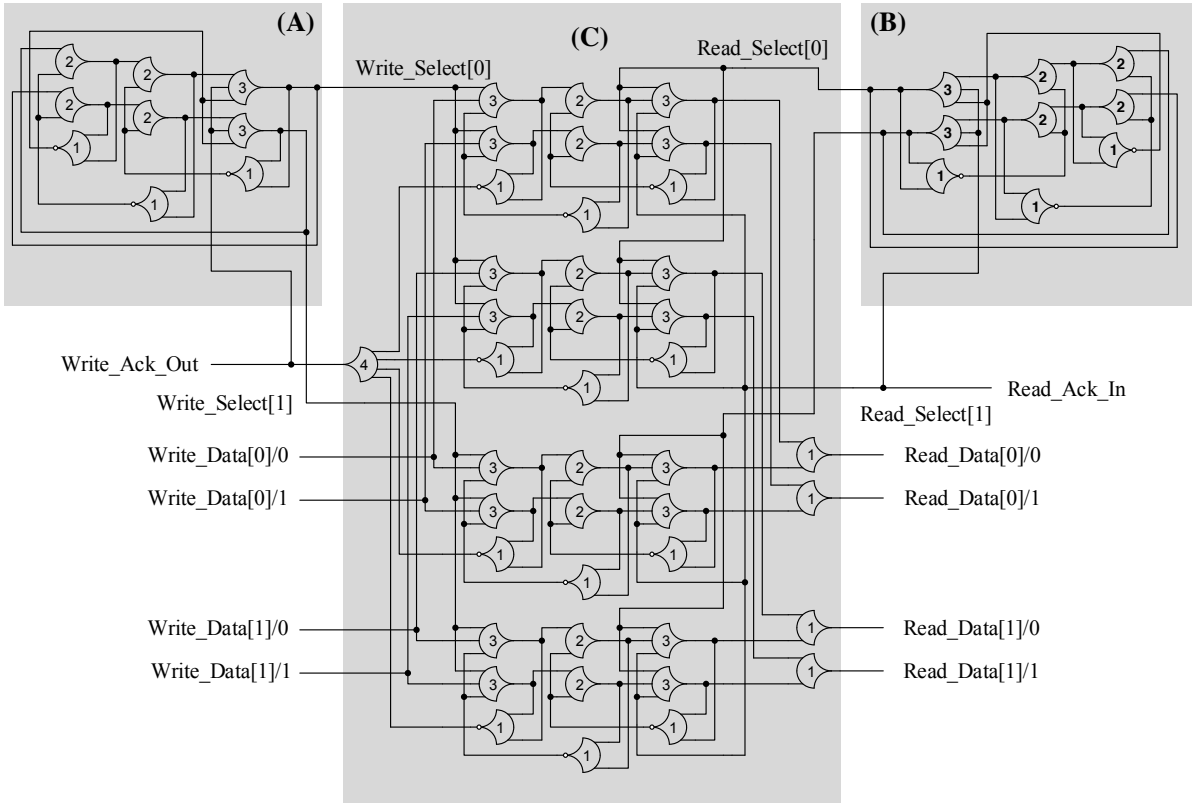

Figure 117: 2x2 NCL FIFO

The example dual-rail encoded NCL FIFO (Figure 117) is two bits wide and two deep. Block (A) shows the Write-Pointer which comprises a two stage state sequencer [18]. The output of this sequencer enables a specific register for writing. Block (C) is a memory (register)

block which contains the data to be written. This 2x2 dual-rail FIFO can hold 8 bits of information. Block (B) is the Read-Pointer. In the same way as the Write-Pointer the output signal of this block points to the register to read. The Write and Read Pointers are controlled using the input and output handshaking signals (Write_Ack_Out and Read_Ack_In).

NCL Write-Back Queue Results Analysis and Comparison

This section describes the comparative results for area, performance and power for each Write-Back Queue type. As previously, these simulations were performed using 28nm UTBB-FDSOI device models with a supply set to 1V. The circuits were designed using NELL, imported to Cadence Virtuoso, and simulated using Cadence Ultrasim and NC-Verilog. The results were obtained for pre-layout, transistor-level simulation.

- Area comparison result and analysis

  Table 18 and Figure 118 present the comparative results between the area of the Data-Queue and the FIFO for the 8-bit Buffer. The Cell (Gates) figures represent the NCL Threshold gate count and area is based on an estimate of the transistor sizes of each gate normalized to a simple inverter of area 1. The Data-Queue uses a much smaller area compared to the FIFO even though it contains both DATA and NULL values. As expected in this pre-layout simulation, area is almost proportional to gate count.

- Performance comparison result and analysis

  Table 19 and Figure 119 show the performance comparison results. When the depth of the buffer is small, the input to output delay of the Data-Queue is smaller than that of the FIFO. Delay increases with depth and exceeds the FIFO delay for depths greater than 5. We note that the input to output delay of deep FIFOs (e.g., depth = 100) makes it unacceptable as a Write-Back buffer. Propagation delay is particularly important in the case where the Register File Write-Back comes quickly after an Operand Register value is read. If the delay of the Buffer is greater than the execution time, it will cause a pipeline stall. The Cycle Time of the Data-Queue is faster than that of the FIFO but if the delay is smaller than the shortest cycle time of the CPU then this will not be a problem. The shortest instruction cycle usually occurs for the NOP (No Operation) instruction but the CPU cycle time of asynchronous processors vary widely with input data, and fast cycles can result from data patterns that cause short carry propagation paths within the ALU.

Compare Type	*Depth x Width*	*Data Queue*	*FIFO*
Cells(Gates)	3x8	130	248
	5x8	234	392
	10x8	494	798
	100x8	5174	8290
Area	3x8	477.50	1107.50
	5x8	859.50	1822.50
	10x8	1814.50	3679.00
	100x8	19004.50	37372.00

Table 18: Data Queue Area Comparison Table

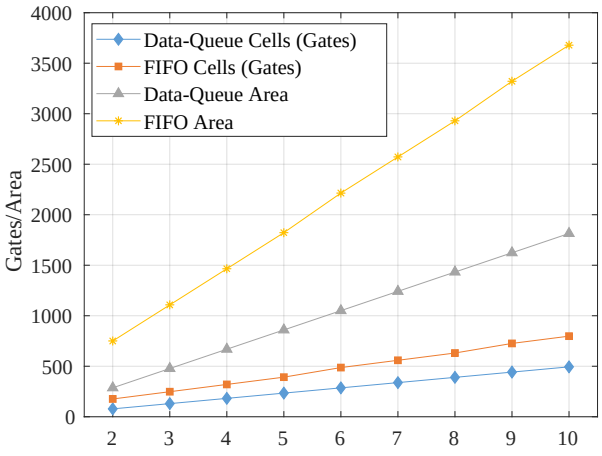

Figure 118: Data Queue Area Comparison Chart

Compare Type	*Depth x Width*	*Data Queue*	*FIFO*
Input to Output Delay	3x8	240ps	469ps
	5x8	437ps	512ps
	10x8	912ps	591ps
	100x8	9,588ps	869ps
Cycle Time	3x8	505ps	648ps
	5x8	506ps	712ps
	10x8	514ps	889ps
	100x8	514ps	1,435ps

Table 19: Data Queue Performance Comparison Table

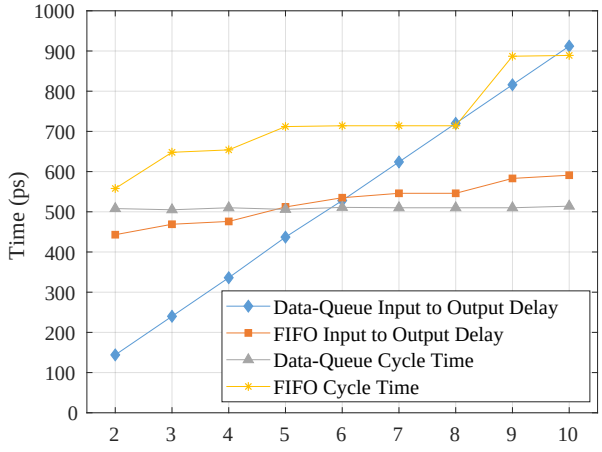

Figure 119: Data Queue Performance Comparison Chart

Compare Type	*Depth x Width*	*Data Queue*	*FIFO*
Average Power Consumption	3x8	17.45uW	15.80uW
	5x8	28.38uW	18.39uW
	10x8	61.19uW	23.79uW
	100x8	653,35uW	103.02uW

Table 20: Data Queue Power Comparison Table

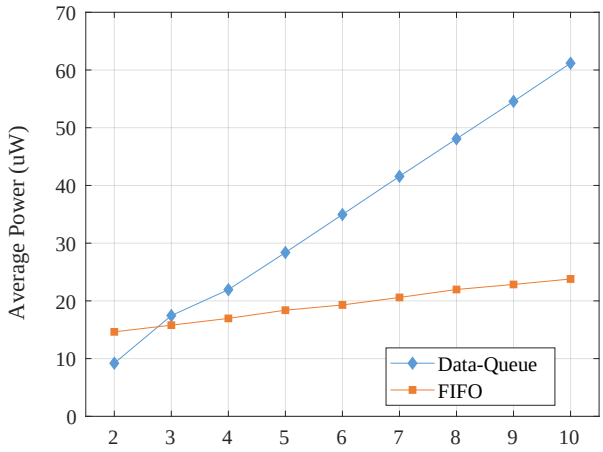

Figure 120: Data Queue Power Comparison Chart

- Power comparison result and analysis

  Table 20 and Figure 120 show the comparison results for power consumption. The average power was measured with a 100MHz (10nS) input transition rate applied under the same conditions to each architecture. As mentioned above, the Write-back Unit will be one of the most active modules in the NCL based RISC CPU (Chapter 5). Therefore, the issue of power consumption is important for this buffer design and exhibits a standard area-power trade-off. The NCL FIFO uses more area but it is more power efficient compared to the Data-Queue when the buffer depth is greater than 3. As the state sequencers control the enable signals of the buffer inputs and outputs, in the memory block (C) of Figure 117, only the selected register switches during each instruction cycle, resulting in increased power efficiency. Both of the buffers (Data-Queue and FIFO) are controlled using the input and output Acknowledge signals. The depth of the Buffer depends on the depth of the pipelining in the Execution Units such as ALU, Multiplier and Floating Point Processing Unit. For example, a floating point multiplier will require deeper queues–in the order of 15. In this case, the data-queue circuit will be unsuitable for that purpose if the system is to maintain high performance (e.g., overall cycle times below 1nS).

From the PPA analysis of these NCL FIFO and data queue [134] structures, the data queue can be seen to offer advantages in area and performance at the expense of input to output delay. Unfortunately, the data queue usually has a small power disadvantages because the input data flows through to the end of the buffer each time. Notwithstanding, a data queue structure was used in this work for the small Write-Back Queue design due to its area/performance advantages.

4.4.4 NCL Completion Tree

In Figure 113, there are three separate blocks: the pair of dual-rail 32:1 OR-Trees (RS1/RS2), the 32-bit x 31 Write Completion Tree and the 32-bit x 9 Read Completion tree. The OR-trees are identical to their synchronous counterparts but in NCL the dual-rail requirement causes them to be twice the size. The write and read completion trees are not present in synchronous designs, of course, but are only required for asynchronous handshaking and so represents a definitive structure that separates the two styles. This issue was raised previously in Chapter 1. The key question is whether the additional completion tree delay due to the wider

32-bit data paths (compared to earlier 8-bit designs) will sufficiently impact the performance of the CPU to cause problems.

Figure 121 shows a number of alternative NCL 32-Bit completion trees. Model (A) is the typical completion logic proposed previously by Fant [18] and Smith [19]. Models (B), (C) and (D) are some alternatives proposed and explored in this work. The models are logically identical.

Model	(A)	(B)	(C)	(D)
Gate Count	43	53	53	21
Transistor Count	404	412	310	388
Logic Depth	4	4	4	3
Expected Delay (ps)	199.7	174.7	164	182

Table 21: NCL Completion Tree Comparison

As can be seen in Table 21, Model (D) is the simplest, being designed with THCOMP and TH44 gates, and results in the fewest net connections. However, Model (C) is the smallest and fastest completion tree design. The two intermediate levels are entirely Boolean and can therefore exploit optimised standard cell libraries for high performance and lower transistor count. Note that in Table 21, the expected delay values were calculated using the average Spice simulation delay of each gate. These huge completion detection trees will greatly increase chip area especially in the register file design but are unavoidable when using this delay insensitive NCL technique.

4.5 The NCL Program Counter Design

The Program Counter (PC) is one of the core machine registers and holds the next instruction fetch address. The address is automatically increased every cycle except for branch instructions and sub-routine calls. The RISC-V ISA does not include special registers, and the PC is not visible to the programmer. Therefore, in our case the PC is not a readable register and only generates the next instruction fetch address.

4.5.1 NCL Program Counter Organization

There are two possible Program Counter styles in NCL. One uses a state machine approach while the other is based on a *ring oscillator*. In this RISC-V design, the ring approach was used for the PC[5]. Figure 122 shows the PC ring oscillator organization. It comprises three

[5]The details of NCL Ring behavior is explained in [18], Chapter 12.

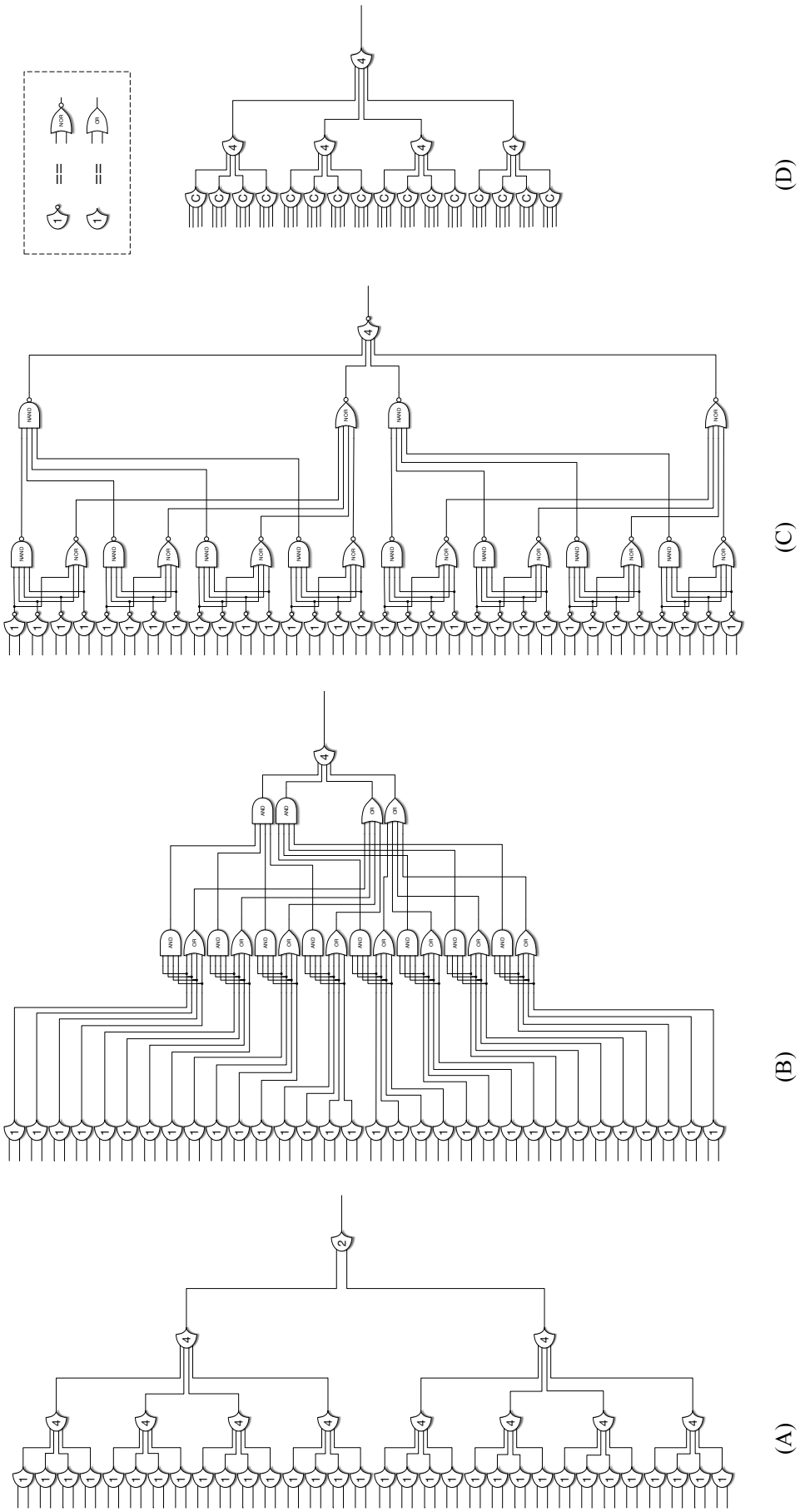

Figure 121: 32-Bit NCL Completion Tree

stages called midPC, curPC and newPC, and oscillates without other external inputs. In the figure, the symbol "R" is the register block and "C" is the completion detection logic shown previously in Figure 121. The basic principle is same as the inverter ring oscillator sometimes found in synchronous designs (Figure 4) and often used as a drop-in test structure in IC fabrication.

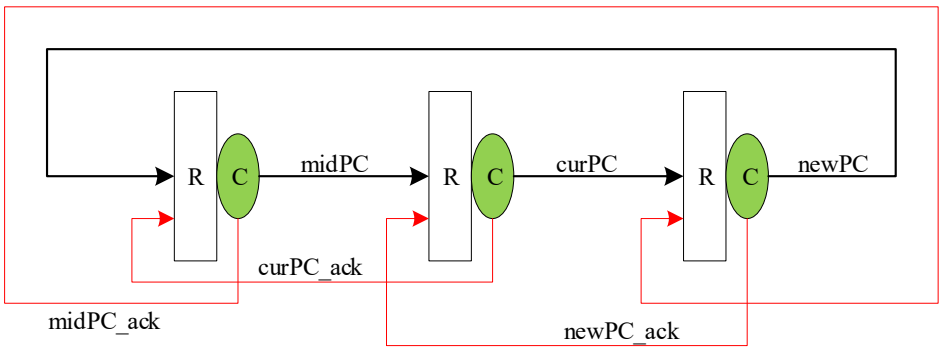

Figure 122: Program Counter Ring

One major difference here is that, in NCL, the ring has to have at least one additional *bubble cycle* otherwise it will not oscillate with sequential Data and NULL cycles. Thus the first element of the ring is a 3-Cycle structure. This scheme obviously does not support the same level of accuracy and stability as a crystal oscillator and its frequency will shift due to PVT variations. In a delay insensitive asynchronous design this is not typically an issue as the system timing is self-adjusting and an accurate execution cycle is not actually required.

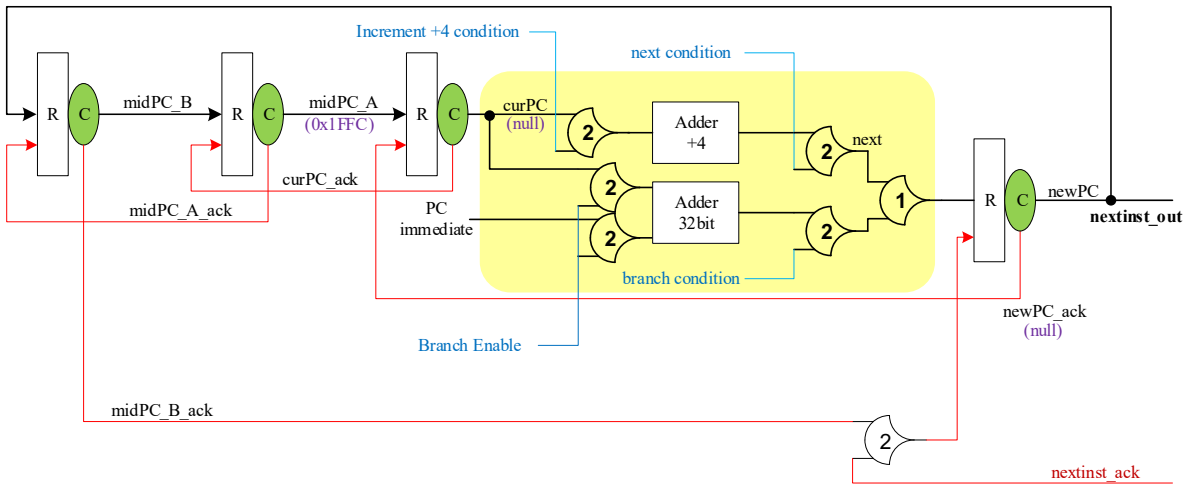

Figure 123: Program Counter Block Diagram

Figure 123 is the block diagram of the Program Counter design. The diagram shows a 4-Cycle ring (midPC_A, midPC_B, curPC and newPC) along with some additional details of

the incrementer (Adder + 4) and the 32-bit addition used for branch operations. The PC also generates signals to the Program Memory interface. The width of the PC ring depends on the Program Memory address width. In this work, a 20-Bit counter was built to support a 4M-Byte Program Memory size.

Register initialization is a very important consideration when starting the RISC-V PC. In Figure 123, there are two NULL initialization points (labeled (null) on newPC_ack Acknowledge and curPC). Also, midPC_A is initialized to 0x1FFC in order to correctly generate address 0x2000 for the RISC-V Reset Vector at the output of the PC incrementer.

4.5.2 NCL State-Machine Design

A state-machine will be the core control mechanism in the control-flow design style used for the RISC-V development. In NCL, a state machine is generated by a combination of 1-bit rings. In this case, only a one 1-bit ring is selected at a time using the transition control signals and their conditional signals. Figure 124 explains the standard NCL control logic using a state machine. Note that the control signals are not dual-rail but are single rail One-Hot control signals. This state machine can be used with the instruction decoder and program counter in the same way as a clocked Boolean logic system.

4.6 Summary

To implement an asynchronous CPU, there are two important parts: the architectural design approach and the asynchronous arithmetic components such as adders, shifters, multipliers and so on. In the synchronous case, the synthesis tools efficiently support most of the arithmetic functions therefore a designer does not need to know the explicit details of the arithmetic components. In contrast, asynchronous technology especially NCL, currently requires the arithmetic components to be designed manually using the NCL library cells, and with limited tool support. In this chapter, the details of the computer arithmetic component designs have been presented and it has been shown how each was designed and optimized. As NCL supports some specific functions that synchronous design does not, such as 2D fine-grained pipelined architectures, the trade-offs implicit in this approach have been discussed and compared. Before composing the CPU core, it is necessary to complete and simulate all of the components plus the sub-module designs for the core, which will be described in the next chapter.

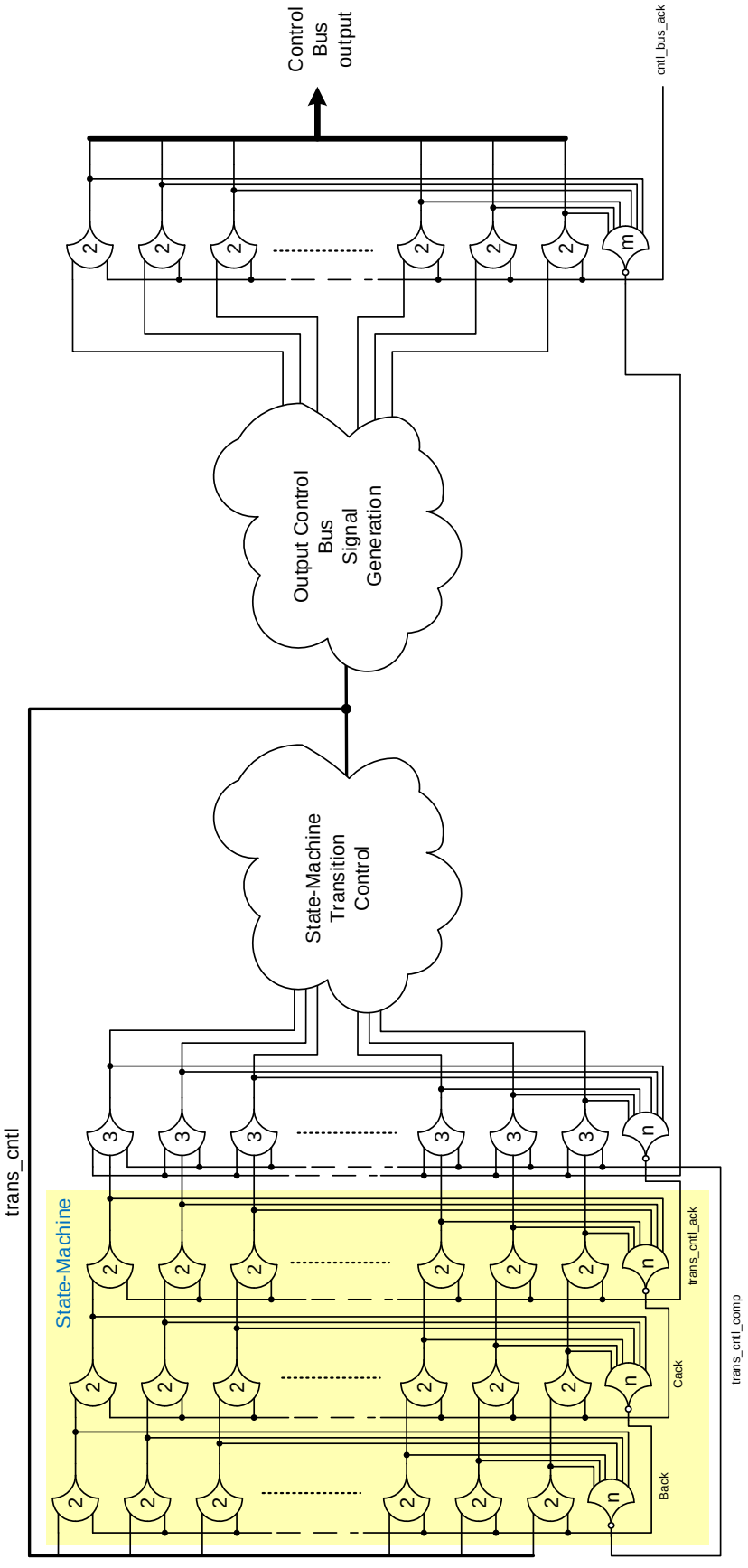

Figure 124: NCL State Machine

Chapter 5

Redback RISC Design and Optimization

This chapter describes the details of the *Redback RISC* core and its power, performance, area results compared to a small number of approximately equivalent synchronous RISC-V cores. As outlined above, the Redback RISC is an asynchronous RISC-V core that has been designed using *Null Convention Logic* technology. The core has a 32-Bit bus architecture and implements the RV32IM instruction set. Just as for previous asynchronous CPU cores, Redback requires optimization approaches that are quite different to the conventional synchronous case. This chapter describes some important optimization techniques that have been used for the core. The RISC-V ISA is briefly introduced in section 2.5 before an explanation of the details of the RISC-V ISA, highlighting which instruction set has been implemented. This chapter also includes details of the methodology that was used for the CPU core comparisons and also the test and verification methodologies.

5.1 RISC-V ISA

RISC-V Instruction Set Architecture (ISA) is the fifth generation Reduced Instruction Set Computer (RISC) Architecture from the University of California Berkeley, which was introduced by Prof. David Patterson and his student Andrew Waterman in 2011 [1]. In [135], the RISC-V ISA is described as a modular design rather than an incremental ISA and identifies its seven goals as: Cost, Simplicity, Performance, Isolation of architecture from implementation, Room for growth, Program size and Ease of programming, compiling and linking.

5.1.1 RISC-V Instructions

RISC-V is an open Instruction Set Architecture (ISA) and the ISA is published and managed by RISC-V Foundation [136]. RISC-V is a modular Instruction Set Architecture and Table 22 shows the extensions list.

[1] Their RISC-V philosophy is encapsulated by their use of the *Mona Lisa* as their book cover and quote *Simplicity is the ultimate sophistication* from Leonardo da Vinci as the answer to "Why RISC-V?"

Table 22: RISC-V ISA Module List

I	Integer ISA - RV32E, RV32I, RV64I, RV128I
M	Integer Multiply/Divide
A	Atomic memory operations (AMOs + LR/SC)
F	Single-precision floating-point
D	Double-precision floating-point
G	IMAFD - General Purpose ISA
Q	Quad-precision floating point
C	Compressed instructions
V	Vector instructions for the data-level parallelism

RV32I is the 32-Bit Base Integer ISA that runs a full software stack. The RV32I is now frozen and will never be changed [135]. The G-extension is for the General Purpose ISA which includes I/M/A/F/D and used for most CPU designs. The RISC-V ISA also supports the C-extension (Compressed Instruction Set) which is a 16-Bit short format with a similar intent to the ARM Thumb Set. The RV32E is the base format with the integer register count reduced to 16 and is designed for embedded systems.

Table 23 shows the RISC-V RV32I base instruction format, which comprises six types: R-type for Register-Register operations, I-type for short Immediate and Loads, S-type for Stores, B-type for Conditional Branches. U-type for Long Immediate and finally, J-type for Unconditional Jumps.

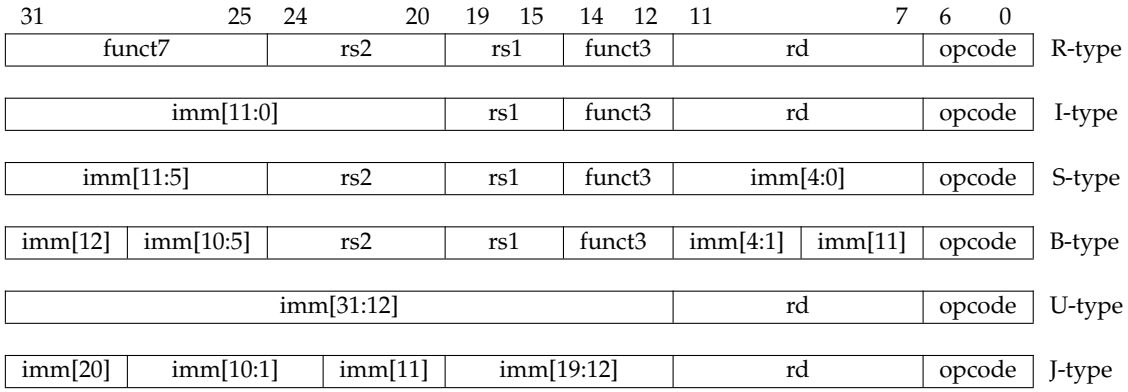

Table 23: RV32I base instruction formats

Table 24 shows the RV32IM Base Instruction Opcode Map that was implemented on the Redback RISC. As shown at the bottom of the Table 24, the RV32M extension module was added to support multiplication and division instructions as these are required by most of the benchmark test programs. The connection between the multiplier and the destination register (RD) is 32-Bit so that the upper 32-Bit and lower 32-bit will be selected from the MULH and

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

Table 24: RV32IM Instruction Opcode Map

MUL instructions respectively. The multiplier and divider need to support both signed and unsigned formats at their input and output interfaces. The instruction details including other Instruction extension modules are described in the RISC-V specification paper [136].

5.1.2 RISC-V Instructions in Asynchronous CPU Design

As introduced above, the goal of the RISC-V open instruction set architecture is simplification. Compared to previous ISAs such as MIPS or ARM, RISC-V is much simpler but has similar code density, fewer condition codes and branch delay slots. The RISC-V instruction set can lead to more straightforward execution control and is well suited to asynchronous implementation as it matches more closely the data-flow behavior of an asynchronous CPU. In addition, as an open ISA, RISC-V is supported by a rapidly evolving "eco-system" of tools and design resources, particularly compiler environments from the open source community, along with academic and commercial RTL implementations that will be available for comparison with our asynchronous CPU core.

5.2 Redback RISC Core Design

In this section, the details of the Redback RISC will be explained. The Redback[2] is, to the best of our knowledge, the first NCL-based asynchronous CPU core based on this ISA.

Most of the NCL-based asynchronous CPU components presented in Chapter 4 are used in here to form the overall CPU core. As mentioned in the previous section, the core is an implementation of RV32IM instruction set and has a fully 32-bit bus architecture. Instruction and data memories are not included in the core as the SRAM memories are not considered as part of CPU core in this design. This core implements the 45 instructions outlined previously in Table 24 as those instructions are the minimum required to be able to successfully run the benchmark tests. This section also includes the core design verification and FPGA prototyping methodologies and their results.

5.2.1 Introduction to Redback RISC

The Redback RISC core has seven major sub-blocks: Program Counter, Instruction Decoder, Arithmetic Logic Unit, Load and Store Unit, Branch Unit, Register File and Multiplier. As shown in Figure 125, the Program Counter and Instruction Decoder have interfaces to the

[2]A preliminary version of this work has been published as "Aristotle. A Logically Determined (Clockless) RISC-V RV32I", 2nd RISC-V Workshop, June 29-30, 2015 [5]

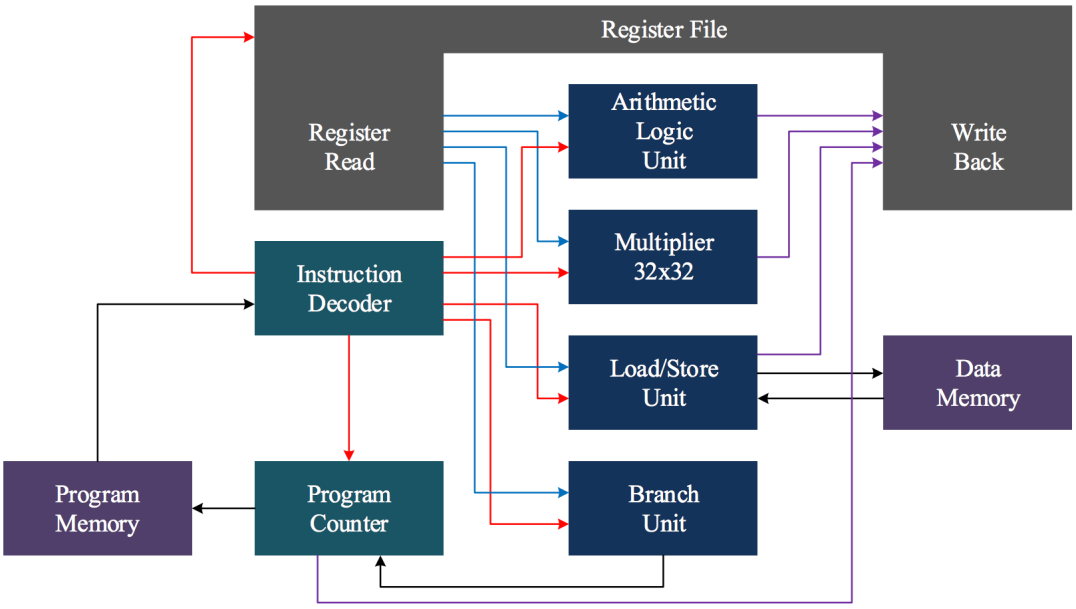

Figure 125: Micro Architecture of Redback RISC

Program Memory, and the Load and Store Unit is connected to the Data Memory. As there are no instruction or data caches, the cache controllers are not implemented in this core.

The design was built using the NELL hardware description language and compiled using NELL compiler. While NELL also supports its own simulator, this was not used for this design. The resulting net-list was simulated using a number of commercial Verilog simulation tools such as Synopsys VCS, Cadence NC-Sim and Mentor Graphics QuestaSim with System Verilog test-bench files. The test vector files for each sub-module were manually generated on an Excel spread sheet based on the RISC-V specification and interface requirements and saved to a *.csv file then converted to *.hex file for the digital simulation tools. Figure 125 shows the Micro Architecture of the Redback RISC. It can be seen that its basic architecture is not significantly different from a conventional synchronous RISC-V CPU core.

Figure 126 illustrates the connections between each sub-module. The connections will be dual-rail or one-hot (single-rail) signals and all the connections have their own acknowledge signals to maintain well controlled data transfer.

5.2.2 Program Counter

The Program Counter has a three- or four-Cycle oscillation ring internally and this ring generates the next instruction fetch address on every cycle. Normally, the next sequential address (current address+4) is produced except in the case where branch or jump conditions

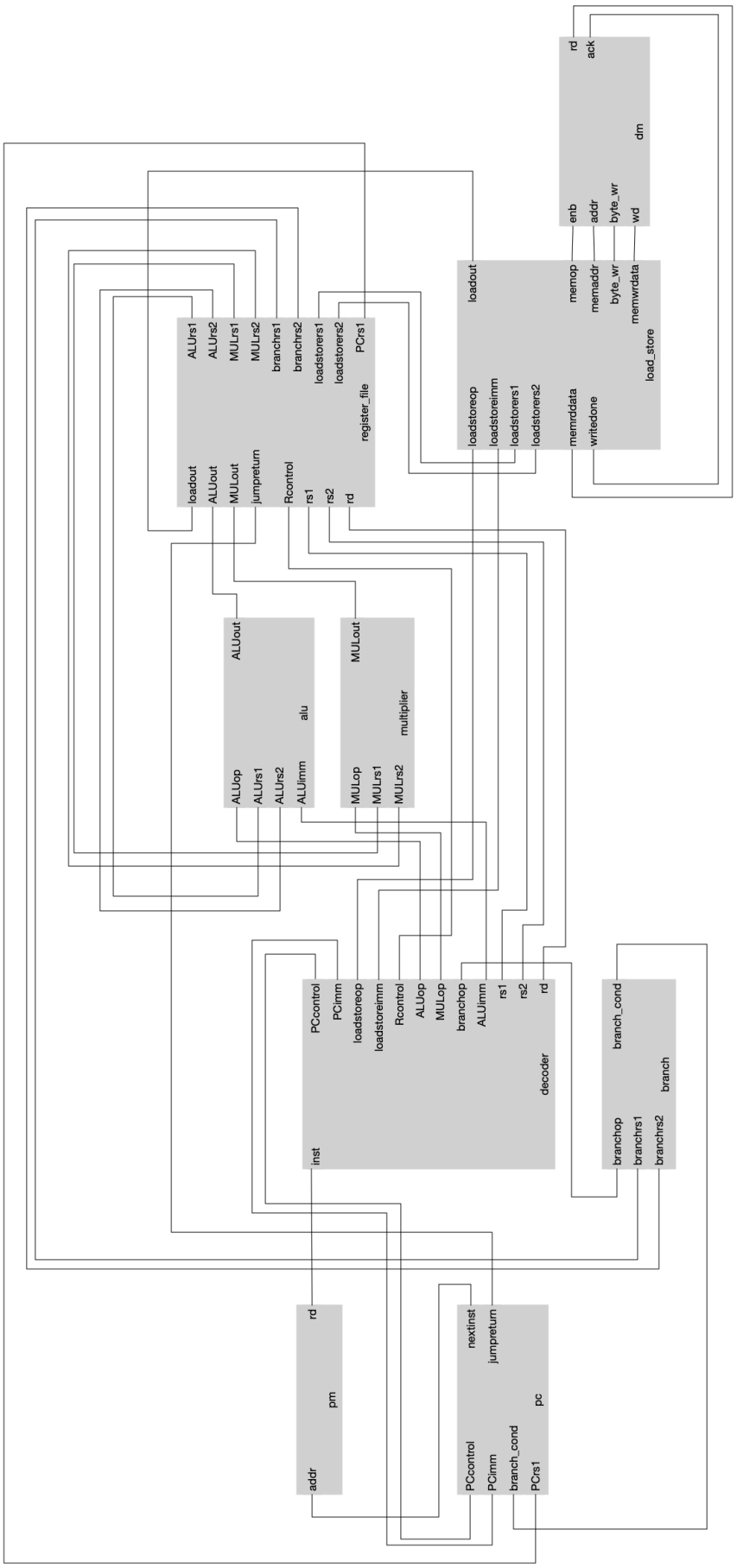

Figure 126: Redback RISC Bus Connection Diagram

are enabled. During the Reset state, the next address is initialized to the value of the CPU reset vector (0x2000 in this RISC-V case).

The oscillator circuit gives *"liveness"* to the CPU so that it operates without the external crystal oscillator required by synchronous machines. It can be controlled (disabled) when the CPU enters Sleep or Power-Save modes and can be woken up by an external interrupt signal.

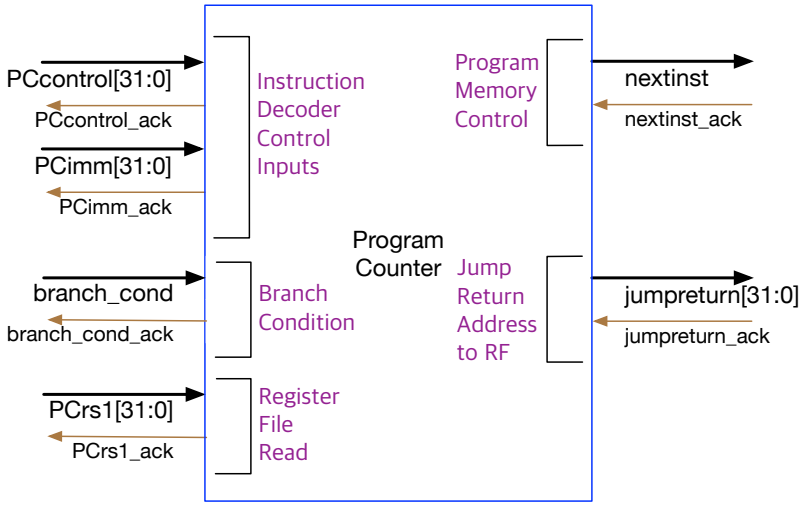

Figure 127: Program Counter Interface Diagram

To control branch execution, the Program Counter receives the branch or jump control signals from the instruction decoder as well as the Boolean result (true/false) from the Branch Unit. When the instruction is JAL, JALR or AUIPC, the jump value is returned to the Register File through the *jumpreturn* bus. The JALR instruction requires the Register File read value (PCrs1) which is then used as the jump address.

5.2.3 Instruction Decoder

The Instruction Decoder decodes the (32-bit) instruction sourced from the Program Memory controller and generates one-hot execution enable signals for the next execution stages. Control signals are derived for the Program Counter, Register File, ALU, Multiplier, Load Store Unit and Branch Unit along with immediate values when called for by specific instructions.

Program Counter control signals:

The Instruction Decoder module generates control signals for the Program Counter after checking whether the instruction involves a branch or unconditional jump;

next - Program Counter increments to the next instruction in most cases except the following

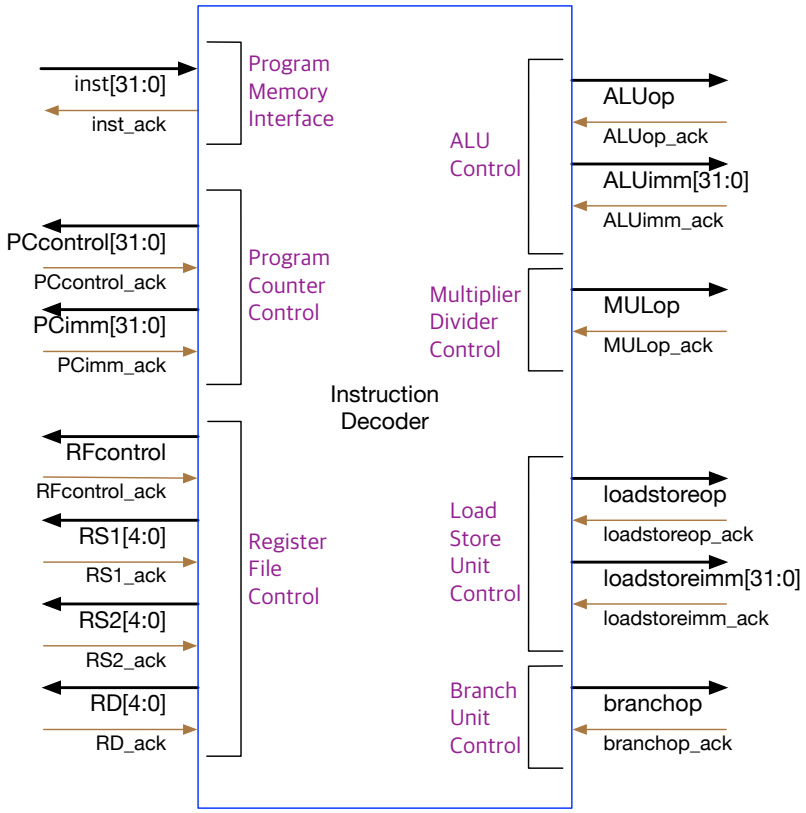

Figure 128: Instruction Decoder Interface Diagram

four:

branch - Program Counter changes to the branch address when the Branch is active;

JALR - Enabled in case of JALR (Jump and Link Register) instruction - Unconditional Jump;

JAL - Enabled in case of JAL (Jump and Link) instruction - Unconditional Jump;

AUIPC - Enabled in case of AUIPC (Add Upper Immediate to PC) to build PC-relative address.

Register File control signals:

The Register File has five read output selection groups (ALU, MUL, Branch Unit, Load Store Unit, Program Counter) and four write-back input selection groups (Load, ALU, MUL, Jump Return), as described in Figure 113. The relevant group is selected based on the input instruction during each cycle. The Instruction Decoder also generates the Register File Source and Destination Address signals based on the input instruction.

ALU - Non-immediate Arithmetic Logic Unit instructions;

ALUimm - Arithmetic Logic Unit instructions with Immediate values;

branch - when branch is active;

PCJALR - JALR instruction (JALR instruction uses rs1);
PCjump - AUIPC, JAL instruction cases (write only), write the calculated Next Address values to the destination register (rd);
LUI - LUI (Load Upper Immediate), (write only), write the Upper Immediate value to the destination register (rd);
load - when Load is active;
store - when Store is active;
MUL - Multiplier and Divider instructions.

Load and PCjump - these have only Register File Write but no Read transactions.
Branch, store - only have Register File Read transaction but have no Write transaction.
Others have both Register File Read and Write transactions.

ALU control signals (ALUop):

This control bus has one-hot control signals for all the ALU related instructions. The ALU instructions are as follows:
ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI, ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND.

Multiplier/Divider control signals (MULop):

This control bus includes one-hot control signals for all the Multiplier/Divider related instructions. These are the instructions:
MUL, MULH, MULHSU, MULHU, DIV, DIVU, REM, REMU.

Load Store Unit control signals (loadstoreop):

This control bus has one-hot control signals for all the Load Store Unit related instructions, which are:
LUI, LB, LH, LW, LBU, LHU, SB, SH, SW.

Branch Unit control signals (branchop):

This control bus has one-hot control signals for all the Branch Unit related instructions:
BEQ, BNE, BLT, BGE, BLTU, BGEU.

5.2.4 Arithmetic Logic Unit

The Arithmetic Logic Unit (ALU) executes integer computational instructions, including integer arithmetic, shift, compare and logic operations.

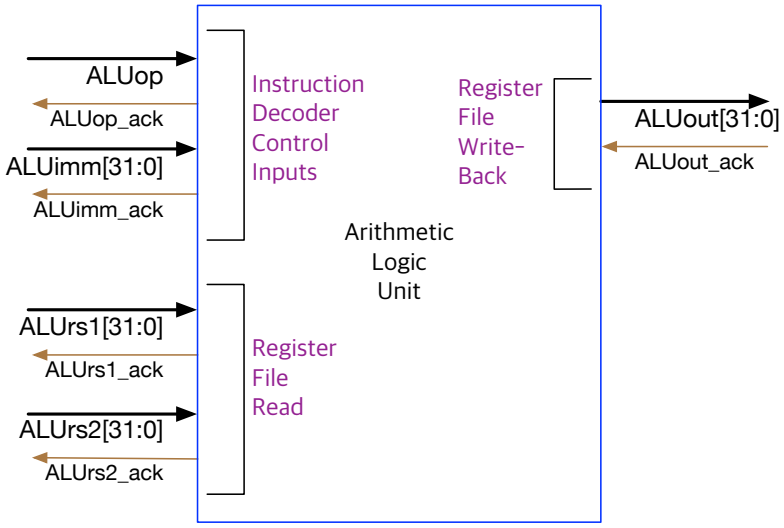

Figure 129: Arithmetic Logic Unit Interface Diagram

The instructions included in this group are:

ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI, ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND.

This NCL based ALU design uses a 32-bit Ripple-Carry Adder and 32-bit Mux-based Barrel Shifter.

5.2.5 Load and Store Unit

The Load and Store Unit interfaces the Instruction Decoder and Register File with Data Memory and controls read and write transactions for the memory. These are the instructions:

LUI, LB, LH, LW, LBU, LHU, SB, SH, SW

memop controls read and write. *byte_wr* control bus has four bit Byte Enable signals.

5.2.6 Branch Unit

Branch Unit interfaces to the Instruction Decoder and Register File and generates the Branch Condition Signals. When this module receives the *branchop* signals from the Instruction Decoder, this module generates the appropriate branch control signals using the two Register File read data for the program counter. The *branch_cond* is the group of one-hot control signals.

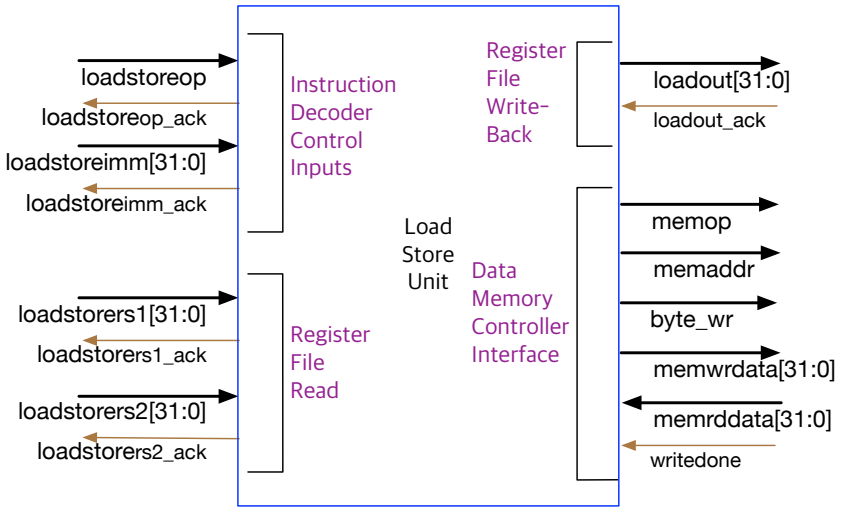

Figure 130: Load Store Unit Interface Diagram

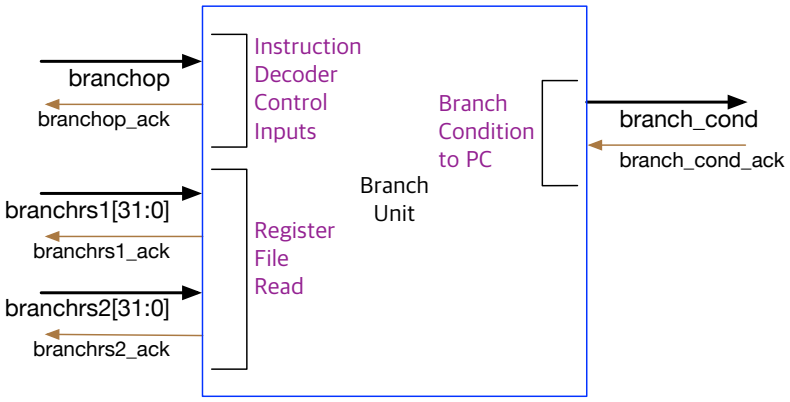

Figure 131: Branch Unit Interface Diagram

Branch condition types:

T - Execute Branch

F - No Branch

The relevant instructions are:

BEQ, BNE, BLT, BGE, BLTU, BGEU.

5.2.7 Register File

In this NCL based RISC-V processor, we are using the RV32IM instruction set which has a 32-Entry, 32-bit Register File (Register[0] is always zero in the RISC-V ISA). The Register File terminates the NCL flow handshaking and the instruction will be completed when the execution results are written back to the Register File write-back ports. In this way, the register File read ports initiate the execution modules. These are termed “Consumer” and “Producer” [18]. The Register File has five read output selection groups (ALU, MUL, Branch Unit, Load

Store Unit, Program Counter) and four write-back input selection groups (Load, ALU, MUL, Jump Return) (Figure 113). During each cycle the appropriate group is selected based on the fetched instruction.

These are the RFcontrol signals:

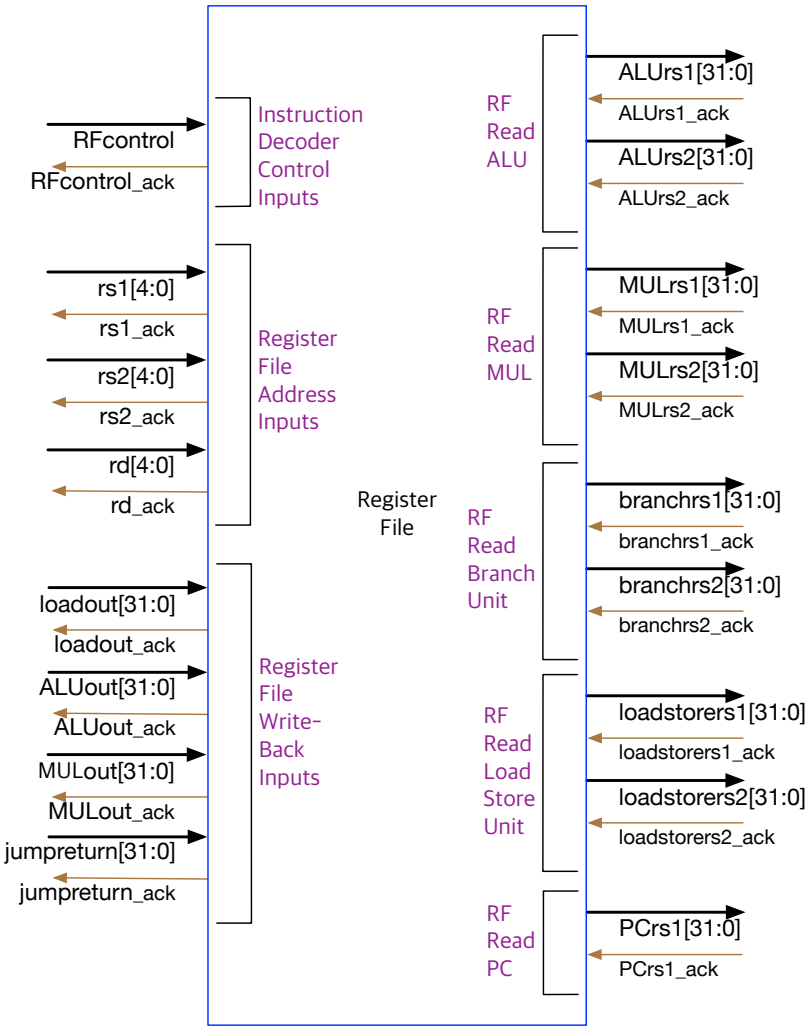

Figure 132: Register File Interface Diagram

ALU - Non-immediate Arithmetic Logic Unit instructions;

ALUimm - Arithmetic Logic Unit; instructions with Immediate values;

branch - when branch is active;

PCJALR - JALR instruction - JALR instruction uses rs1;

PCjump - AUIPC, JAL instruction cases (write only), write the calculated Next Address values to the destination register (rd);

LUI - LUI (Load Upper Immediate), (write only), write the Upper Immediate value to the destination register(rd);

load - when Load is active;

store - when Store is active;

MUL - Multiplier and Divider instructions;

load and PCjump - only have Register File Write transaction but have no Read transaction.

branch, store - only have Register File Read transaction but have no Write transaction.

5.2.8 Multiplier

A 32-bit NCL Modified-Booth multiplier has been implemented for this RISC-V processor [3] that has three pipeline options—1, 3 or 4 stages. The 4-pipeline stage multiplier organization is faster than the others while exhibiting similar area/power numbers so has been the preferred option for the CPU design. (Figure 134).

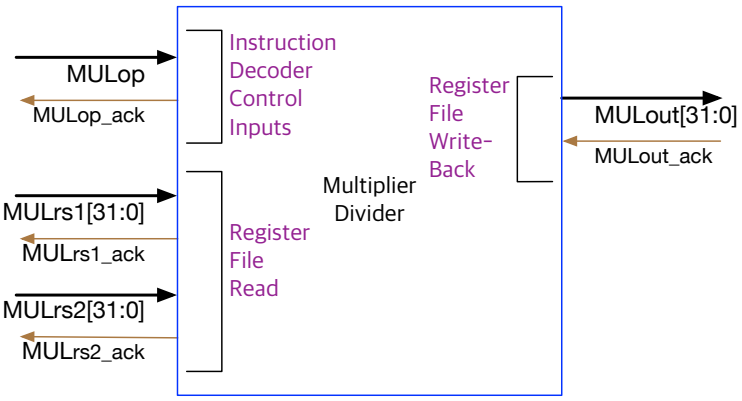

Figure 133: Multiplier Interface Diagram

These are the instructions:

MUL, MULH, MULHSU, MULHU, DIV, DIVU, REM, REMU.

5.2.9 Program Memory and Data Memory Interface

In the conventional way, the Program Memory holds the instruction code generated by the RISC-V compiler and is accessed during the instruction fetch cycle. Program memory is read-only and can be replaced with the Instruction Cache Controller. Figure 135 shows the flow when the Program Counter starts the fetch cycle and the Instruction Decoder receives the fetched instructions. As presented, this diagram presupposes the use of a synchronous memory organization, such as is required by FPGA implementations and that exhibits a fixed Memory

[3] Since RV32IM architecture has only 32-bit Registers, the multiplier is only required for 32-bit output. The Modified-Booth Multiplier output port described in the Chapter-4 has been modified to have only 32-bit output ports and divided into lower 32 bits and upper 32 bits

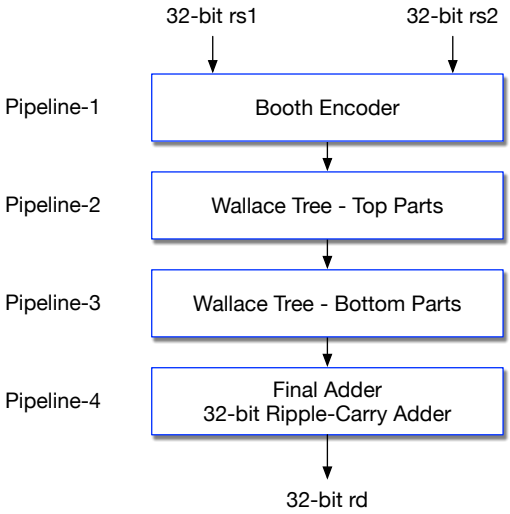

Figure 134: Multiplier Pipeline Diagram

Access Time. However, this can be easily replaced by an asynchronous memory architecture with handshaking interfaces (Request/Acknowledge) with minimal changes to the diagram.

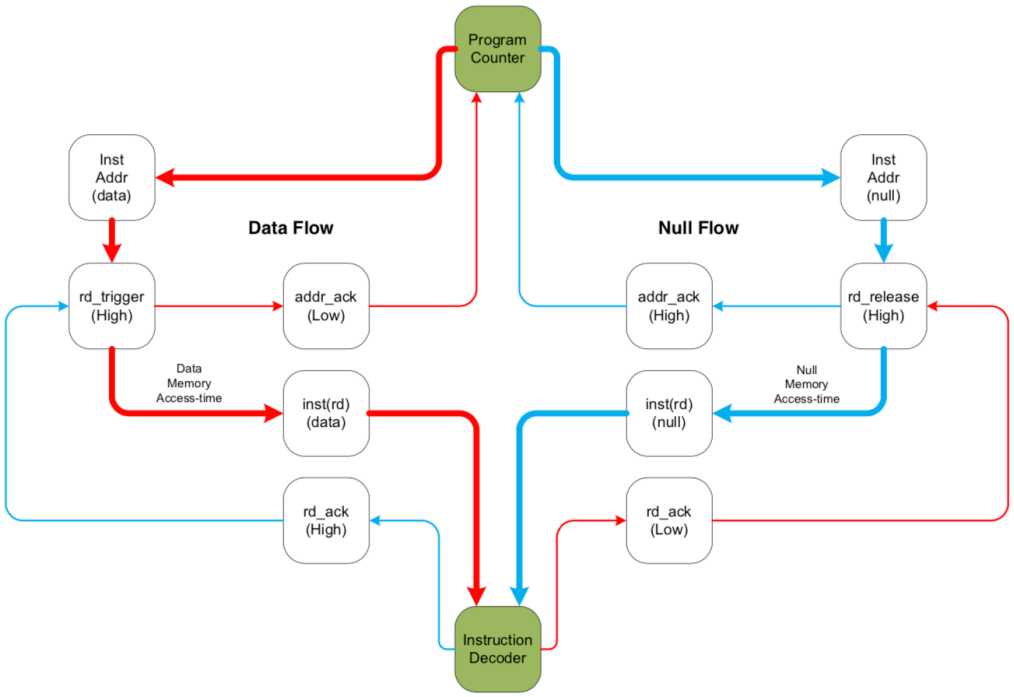

Figure 135: Program Memory Data Flow Diagram

Because the fetch cycle is executed asynchronously, its cycle time cannot be faster than the memory access time (but can be slower). It also requires single-rail to dual-rail conversion and vice versa. In a similar way, the Load Store Unit accesses Data Memory to load or store

data from the Register File. Program Memory only requires read control whereas Data Memory needs to support both read and write functions and therefore requires additional control signals for the write operation.

5.2.10 Summary of Data Flow

Figure 136 summarises the data flow of the whole Redback RISC core. The memory blocks colored yellow are outside of the core, which includes only the memory interfaces.

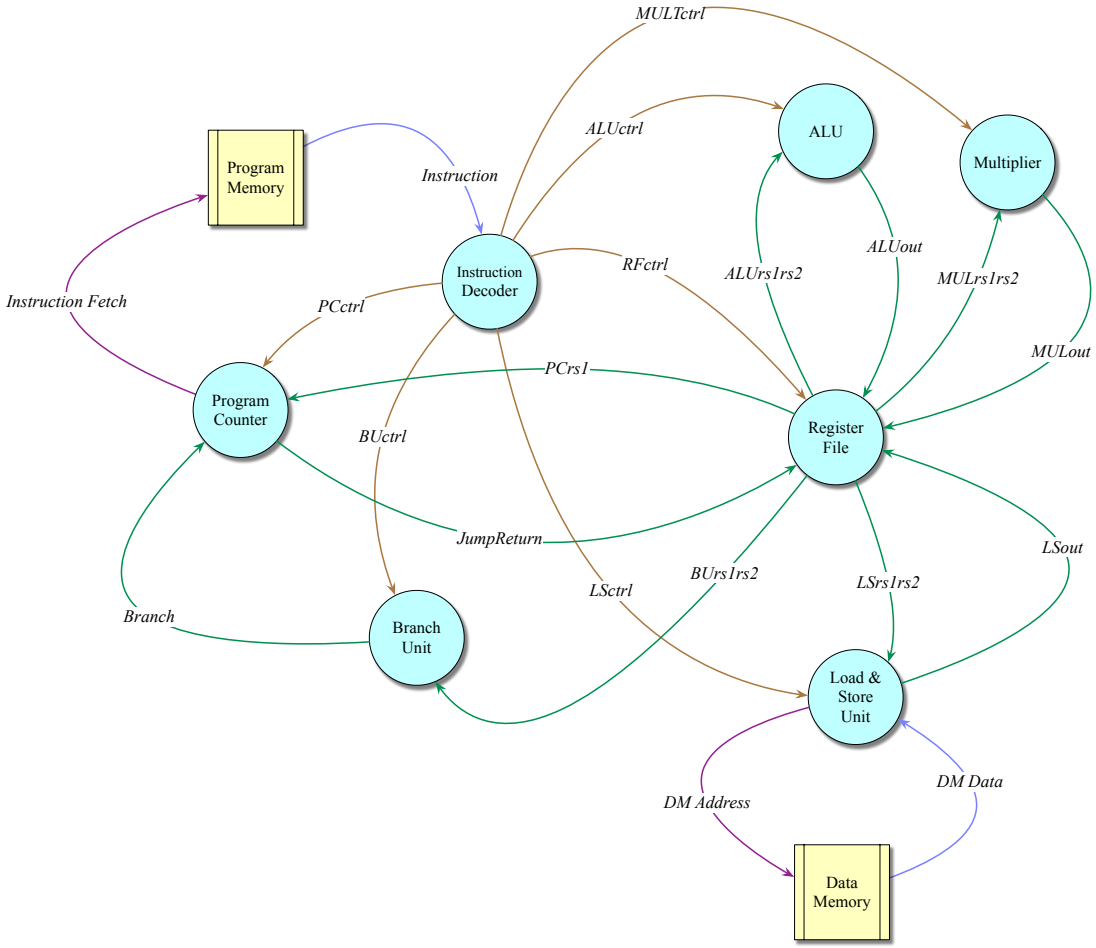

Figure 136: Summary of Data Flow

5.2.11 Verification of Redback RISC core

As mentioned previously, the generated net-list was simulated using commercial Verilog simulation tools such as Synopsys VCS, Cadence NC-Sim and Mentor Graphics QuestaSim with System Verilog test-bench files. The test vector files for each sub-module have been manually generated on an Excel spread sheet based on the RISC-V specification and interface requirements and saved to .csv files, then converted to .hex files for the digital simulation tools. Table

25 shows an example of test-vector generation, in this case for the part of Instruction Decoder. It covers all of the implemented instruction cases and also generates their handshaking signals. The test-vectors also include the expected results for output verification. Some interfaces are dual-rail and some are one-hot single-rail but both types of busses require acknowledge signals.

Once all of the individual sub-module passed their basic verification, the core-level functions were simulated using the RISC-V test programs derived from the *riscv-tests* repository [137]). These test programs are generally used for all the RISC-V CPU core verification, and also to validate the ISA simulator tools. The *riscv-tests* repository includes files for *isa-verification* and *benchmark-tests*. The RV32IM binary image (.hex files) was generated for each ISA test-case using the RISC-V compile environments and functional simulation performed after reading the executable binary images into the Program Memory block. The tests produce Pass/Fail results that can be read from the System Verilog test-bench. The test-bench was built using System Verilog and simulated using the Cadence NC-Verilog simulation tools. An automatic test environment for each instruction was generated using Python scripts plus a Makefile script to avoid a lot of manual simulation.

Table 26 shows the Pass/Fail results of all the simulated RV32UI test-cases. Some of the test cases failed because a number of the specialized instructions such as *Privileged Atomic Instructions* were not implemented in this design. After ISA simulation, the design was tested using the benchmarks from *riscv-tests* [137]. These support 12 different CPU test cases and some additional tests can be added to this list if required. Using these benchmarks, the CPU design was verified and its PPA (Performance/Power/Area) analysed.

Unlike conventional synchronous systems, in NCL we have to use behavioral models to run gate-level simulations for the functional verification phase because the design language NELL used to describe the CPU does not directly support conventional ASIC simulation tools.

5.2.12 Redback RISC Core Prototyping using Commercial FPGA

The Redback RISC core was initially tested on a commercial FPGA board, the Terasic DE4, that uses a Stratix-4GX device (i.e., EP4SGX230KF40C2). As mentioned in Section 3.3, NCL designs can be implemented on the commercial FPGA with some limitations. We have developed an NCL Cell library aimed at specific FPGA vendors such as Xilinx, Altera/Intel

```
// Table of Instruction Decoder test case for test-bench which is tb_decoder.sv
// designed by Matthew Kim - 22/04/2015


//enum {ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI, ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND} ALUop_type;
//enum {FENCE, FENCE_I, SCALL, SBREAK, RDCYCLE, RDCYCLEH, RDTIME, RDTIMEH, RDINSTRET, RDINSTRETH, UNKNOWN} sysop_type;
//enum {BEQ, BNE, BLT, BGE, BLTU, BGEU} branchop_type;
//enum {LUI, LB, LH, LW, LBU, LHU, SB, SH, SW} loadstoreop_type;
//enum {JALR, JAL, AUIPC} jumpop_type;

//enum {next_pc, branch_pc, JAL_pc, JALR_pc, AUIPC_pc} PCcontrol_type;
//enum {ALU3, ALU2, branch_rcntl, PCJALR, PCjump, loadA, loadB, store, sysreg} Rcontrol_type; // Register File Control

Instruction
               ,inst(Binary - Example)
               ,                                      ,inst(Hex - Example, dual-rail, binary)
               ,                                      ,          ,PCcontrol_out - expected(single-rail, one-hot)
               ,                                      ,          ,   ,PCimm_out - expected(dual-rail, binary)
               ,                                      ,          ,   ,          ,loadstoreop_out - expected(single-rail, one-hot)
               ,                                      ,          ,   ,          ,       ,loadstoreimm_out - expected([11:0]valid, dual-rail, binary)
               ,                                      ,          ,   ,          ,       ,          ,Rcontrol_out - expected(single-rail, one-hot)
               ,                                      ,          ,   ,          ,       ,          ,       ,ALUop_out - expected(single-rail, one-hot)
               ,                                      ,          ,   ,          ,       ,          ,       ,       ,branchop_out - expected(single-rail, one-hot)
               ,                                      ,          ,   ,          ,       ,          ,       ,       ,   ,ALUimm_out - expected([11:0]valid, dual-rail, binary)
               ,                                      ,          ,   ,          ,       ,          ,       ,       ,   ,          ,rs1_out - expected(single-rail, one-hot)
               ,                                      ,          ,   ,          ,       ,          ,       ,       ,   ,          ,   ,rs2_out - expected(single-rail, one-hot)
               ,                                      ,          ,   ,          ,       ,          ,       ,       ,   ,          ,   ,   ,rd_out - expected(single-rail, one-hot)
               ,                                      ,          ,   ,          ,       ,          ,       ,       ,   ,          ,   ,   ,   ,Instruction Type

// ALUop_type - ALU2 - aluimm_active
ADDI           ,1100_1010_0101_0000_1000_0001_0001_0011    ,CA508113   ,01 ,xxxxxxxx   ,xxx    ,xxxxxxxx   ,002    ,00001  ,xx ,FFFFFCA5   ,01 ,xx ,02 ,I-type     ,// imm - sign-extended
SLTI           ,1010_0000_1111_0010_0010_0001_1001_0011    ,A0F22193   ,01 ,xxxxxxxx   ,xxx    ,xxxxxxxx   ,002    ,00002  ,xx ,FFFFFA0F   ,04 ,xx ,03 ,I-type     ,// imm - sign-extended
SLTIU          ,1010_0110_1001_0010_1011_0011_0001_0011    ,A692B313   ,01 ,xxxxxxxx   ,xxx    ,xxxxxxxx   ,002    ,00004  ,xx ,FFFFFA69   ,05 ,xx ,06 ,I-type     ,// imm - sign-extended
XORI           ,1100_1010_0101_0000_1100_0001_0001_0011    ,CA50C113   ,01 ,xxxxxxxx   ,xxx    ,xxxxxxxx   ,002    ,00008  ,xx ,FFFFFCA5   ,01 ,xx ,02 ,I-type     ,// imm - sign-extended
ORI            ,0010_0000_1111_0010_0110_0001_1001_0011    ,20F26193   ,01 ,xxxxxxxx   ,xxx    ,xxxxxxxx   ,002    ,00010  ,xx ,0000020F   ,04 ,xx ,03 ,I-type     ,// imm - sign-extended
ANDI           ,0010_0110_1001_0010_1111_0011_0001_0011    ,2692F313   ,01 ,xxxxxxxx   ,xxx    ,xxxxxxxx   ,002    ,00020  ,xx ,00000269   ,05 ,xx ,06 ,I-type     ,// imm - sign-extended
SLLI           ,0000_0000_0101_0001_0001_0000_1001_0011    ,00511093   ,01 ,xxxxxxxx   ,xxx    ,xxxxxxxx   ,002    ,00040  ,xx ,00000005   ,02 ,xx ,01 ,I-type     ,// imm - zero-extended(imm 5bit valid)
SRLI           ,0000_0000_1010_0010_0101_0001_1001_0011    ,00A25193   ,01 ,xxxxxxxx   ,xxx    ,xxxxxxxx   ,002    ,00080  ,xx ,0000000A   ,04 ,xx ,03 ,I-type     ,// imm - zero-extended(imm 5bit valid)
SRAI           ,0100_0001_1111_1111_1101_0011_1001_0011    ,41FFD393   ,01 ,xxxxxxxx   ,xxx    ,xxxxxxxx   ,002    ,00100  ,xx ,0000001F   ,1F ,xx ,07 ,I-type     ,// imm - zero-extended(imm 5bit valid)
// ALUop_type - ALU3 - alu_active
ADD            ,0000_0000_0100_1111_1000_0000_1011_0011    ,004F80B3   ,01 ,xxxxxxxx   ,xxx    ,xxxxxxxx   ,001    ,00200  ,xx ,xxxxxxxx   ,1F ,04 ,01 ,R-type     ,
SUB            ,0100_0000_0101_1111_0000_0001_0011_0011    ,405F0133   ,01 ,xxxxxxxx   ,xxx    ,xxxxxxxx   ,001    ,00400  ,xx ,xxxxxxxx   ,1E ,05 ,02 ,R-type     ,
SLL            ,0000_0000_0110_1110_1001_0001_1011_0011    ,006E91B3   ,01 ,xxxxxxxx   ,xxx    ,xxxxxxxx   ,001    ,00800  ,xx ,xxxxxxxx   ,1D ,06 ,03 ,R-type     ,
SLT            ,0000_0000_0111_1110_0010_0010_0011_0011    ,007E2233   ,01 ,xxxxxxxx   ,xxx    ,xxxxxxxx   ,001    ,01000  ,xx ,xxxxxxxx   ,1C ,07 ,04 ,R-type     ,
SLTU           ,0000_0000_1000_1101_1011_0010_1011_0011    ,008DB2B3   ,01 ,xxxxxxxx   ,xxx    ,xxxxxxxx   ,001    ,02000  ,xx ,xxxxxxxx   ,1B ,08 ,05 ,R-type     ,
XOR            ,0000_0000_1001_1101_0100_0011_0011_0011    ,009D4333   ,01 ,xxxxxxxx   ,xxx    ,xxxxxxxx   ,001    ,04000  ,xx ,xxxxxxxx   ,1A ,09 ,06 ,R-type     ,
SRL            ,0000_0000_1010_1100_1101_0011_1011_0011    ,00ACD3B3   ,01 ,xxxxxxxx   ,xxx    ,xxxxxxxx   ,001    ,08000  ,xx ,xxxxxxxx   ,19 ,0A ,07 ,R-type     ,
SRA            ,0100_0000_1011_1100_0101_0100_0011_0011    ,40BC5433   ,01 ,xxxxxxxx   ,xxx    ,xxxxxxxx   ,001    ,10000  ,xx ,xxxxxxxx   ,18 ,0B ,08 ,R-type     ,
OR             ,0000_0000_1100_1011_1110_0100_1011_0011    ,00CBE4B3   ,01 ,xxxxxxxx   ,xxx    ,xxxxxxxx   ,001    ,20000  ,xx ,xxxxxxxx   ,17 ,0C ,09 ,R-type     ,
AND            ,0000_0000_1101_1011_0111_0101_0011_0011    ,00DB7533   ,01 ,xxxxxxxx   ,xxx    ,xxxxxxxx   ,001    ,40000  ,xx ,xxxxxxxx   ,16 ,0D ,0A ,R-type     ,

// branchop_type
BEQ            ,0000_1100_1000_0000_1000_0001_1110_0011    ,0C8081E3   ,02 ,000008C2   ,xxx    ,xxxxxxxx   ,004    ,xxxxx  ,01 ,xxxxxxxx   ,01 ,08 ,xx ,SB-type    ,// imm - sign-extended - Branch Offset
BNE            ,1000_1010_1001_0001_0001_0010_0110_0011    ,8A911263   ,02 ,FFFFF0A4   ,xxx    ,xxxxxxxx   ,004    ,xxxxx  ,02 ,xxxxxxxx   ,02 ,09 ,xx ,SB-type    ,// imm - sign-extended
BLT            ,0000_1000_1010_0001_1100_0011_1110_0011    ,08A1C3E3   ,02 ,00000886   ,xxx    ,xxxxxxxx   ,004    ,xxxxx  ,04 ,xxxxxxxx   ,03 ,0A ,xx ,SB-type    ,// imm - sign-extended
BGE            ,1000_0110_1011_0010_0101_0100_0110_0011    ,86B25463   ,02 ,FFFFF068   ,xxx    ,xxxxxxxx   ,004    ,xxxxx  ,08 ,xxxxxxxx   ,04 ,0B ,xx ,SB-type    ,// imm - sign-extended
BLTU           ,0000_0100_1100_0010_1110_0101_1110_0011    ,04C2E5E3   ,02 ,0000084A   ,xxx    ,xxxxxxxx   ,004    ,xxxxx  ,10 ,xxxxxxxx   ,05 ,0C ,xx ,SB-type    ,// imm - sign-extended
BGEU           ,1000_0010_1101_0011_0111_0110_0110_0011    ,82D37663   ,02 ,FFFFF02C   ,xxx    ,xxxxxxxx   ,004    ,xxxxx  ,20 ,xxxxxxxx   ,06 ,0D ,xx ,SB-type    ,// imm - sign-extended

// loadstoreop_type - loadA
LUI            ,1111_1110_1101_1100_1011_0000_1011_0111    ,FEDCB0B7   ,01 ,xxxxxxxx   ,001    ,FEDCB000   ,020    ,xxxxx  ,xx ,xxxxxxxx   ,xx ,xx ,01 ,I-type     ,// Low 12 bits fills Zeros
// loadstoreop_type - loadB
LB             ,0000_0001_0000_0010_0000_0001_0000_0011    ,01020103   ,01 ,xxxxxxxx   ,002    ,00000010   ,040    ,xxxxx  ,xx ,xxxxxxxx   ,04 ,xx ,02 ,I-type     ,// imm - sign-extended
LH             ,0000_0010_0000_0100_0001_0010_0000_0011    ,02041203   ,01 ,xxxxxxxx   ,004    ,00000020   ,040    ,xxxxx  ,xx ,xxxxxxxx   ,08 ,xx ,04 ,I-type     ,// imm - sign-extended
LW             ,0000_0100_0000_0110_0010_0011_0000_0011    ,04062303   ,01 ,xxxxxxxx   ,008    ,00000040   ,040    ,xxxxx  ,xx ,xxxxxxxx   ,0C ,xx ,06 ,I-type     ,// imm - sign-extended
LBU            ,0101_0101_0101_1000_0100_0100_0000_0011    ,55584403   ,01 ,xxxxxxxx   ,010    ,00000555   ,040    ,xxxxx  ,xx ,xxxxxxxx   ,10 ,xx ,08 ,I-type     ,// imm - sign-extended
LHU            ,1010_1010_1010_1010_0101_0101_0000_0011    ,AAAA5503   ,01 ,xxxxxxxx   ,020    ,FFFFFAAA   ,040    ,xxxxx  ,xx ,xxxxxxxx   ,14 ,xx ,0A ,I-type     ,// imm - sign-extended
// loadstoreop_type - store
SB             ,0101_0100_0010_0000_1000_1010_1010_0011    ,54208AA3   ,01 ,xxxxxxxx   ,040    ,00000555   ,080    ,xxxxx  ,xx ,xxxxxxxx   ,01 ,02 ,xx ,S-type     ,// imm - sign-extended
SH             ,1010_1010_0100_0001_0001_0101_0010_0011    ,AA411523   ,01 ,xxxxxxxx   ,080    ,FFFFFAAA   ,080    ,xxxxx  ,xx ,xxxxxxxx   ,02 ,04 ,xx ,S-type     ,// imm - sign-extended
SW             .1111 1110 1000 0010 0010 1111 1010 0011    .FE822FA3   .01 .xxxxxxxx   .100    .FFFFFFFF   .080    .xxxxx  .xx .xxxxxxxx   .04 .08 .xx .S-type     .// imm - sign-extended
```

Table 25: Instruction Decoder Test-Case Table

Table 26: ISA Test Results

ISA test	Result	ISA test	Result
rv32ui-p-remu.hex	Pass	rv32ui-p-sb.hex	Pass
rv32ui-p-amoor_w.hex	Fail	rv32ui-p-fence_i.hex	Fail
rv32ui-p-srai.hex	Pass	rv32ui-p-mul.hex	Pass
rv32ui-p-divu.hex	Pass	rv32ui-p-simple.hex	Pass
rv32ui-p-slti.hex	Pass	rv32ui-p-rem.hex	Pass
rv32ui-p-add.hex	Pass	rv32ui-p-j.hex	Pass
rv32ui-p-lh.hex	Pass	rv32ui-p-jalr.hex	Pass
rv32ui-p-mulhsu.hex	Pass	rv32ui-p-slli.hex	Pass
rv32ui-p-sw.hex	Pass	rv32ui-p-mulhu.hex	Pass
rv32ui-p-bne.hex	Pass	rv32ui-p-addi.hex	Pass
rv32ui-p-amomaxu_w.hex	Fail	rv32ui-p-amominu_w.hex	Fail
rv32ui-p-bltu.hex	Pass	rv32ui-p-amomin_w.hex	Fail
rv32ui-p-amoadd_w.hex	Fail	rv32ui-p-sh.hex	Pass
rv32ui-p-amomax_w.hex	Fail	rv32ui-p-or.hex	Pass
rv32ui-p-and.hex	Pass	rv32ui-p-slt.hex	Pass
rv32ui-p-blt.hex	Pass	rv32ui-p-lb.hex	Pass
rv32ui-p-andi.hex	Pass	rv32ui-p-srli.hex	Pass
rv32ui-p-bge.hex	Pass	rv32ui-pm-lrsc.hex	Fail
rv32ui-p-lui.hex	Pass	rv32ui-p-lhu.hex	Pass
rv32ui-p-auipc.hex	Pass	rv32ui-p-sub.hex	Pass
rv32ui-p-beq.hex	Pass	rv32ui-p-amoswap_w.hex	Fail
rv32ui-p-xori.hex	Pass	rv32ui-p-xor.hex	Pass
rv32ui-p-lbu.hex	Pass	rv32ui-p-sra.hex	Pass
rv32ui-p-div.hex	Pass	rv32ui-p-bgeu.hex	Pass
rv32ui-p-lw.hex	Pass	rv32ui-p-ori.hex	Pass
rv32ui-p-srl.hex	Pass	rv32ui-p-amoand_w.hex	Fail
rv32ui-p-jal.hex	Pass	rv32ui-p-mulh.hex	Pass
rv32ui-p-sll.hex	Pass		

and Actel that uses their internal Look Up Tables. Because the commercial FPGA only contains synchronous SRAM memory, it was necessary to insert an asynchronous/synchronous interface circuit on the Program Memory and Data Memory controllers. This FPGA testing process is very inefficient because it only uses LUTs with each NCL gate being mapped onto one LUT cell. Therefore, this technique is used only for verification and prototyping purposes but not for the actual implementation.

Figure 137 and Figure 138 show the SignalTap screen capture. The Dhrystone Benchmark test image (dhrystone.riscv.hex) was executed and Figure 137 shows the Program Counter and Program Memory access waveform and Figure 138 shows the waveforms for the instruction decoder program counter control signals and the register file control signals. As the NCL gates are implemented on LUTs, and the delay time of each is almost same, the waveforms

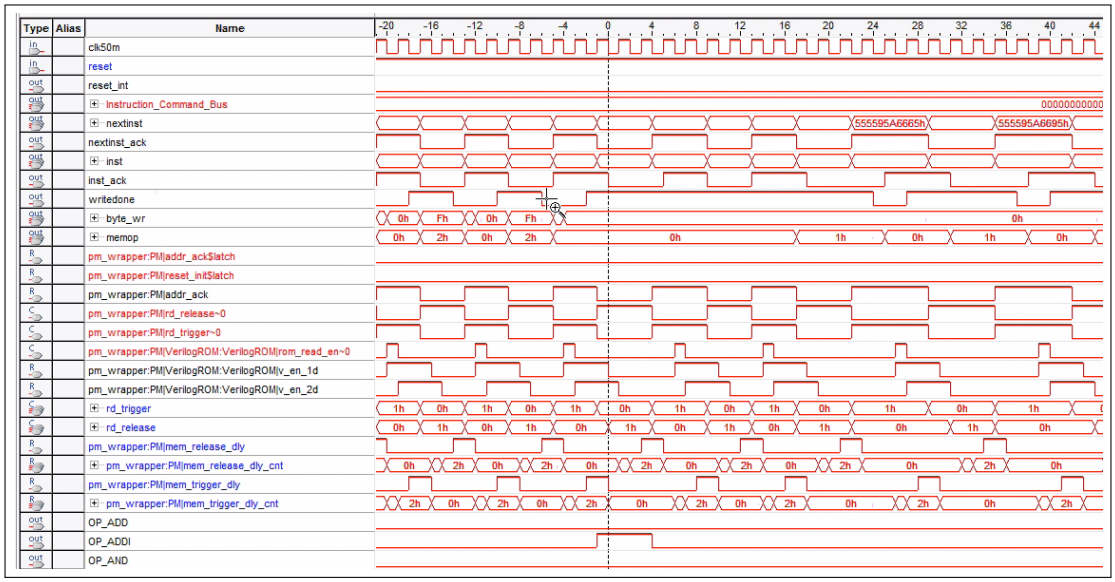

Figure 137: SingalTap FPGA Internal Signal Monitoring

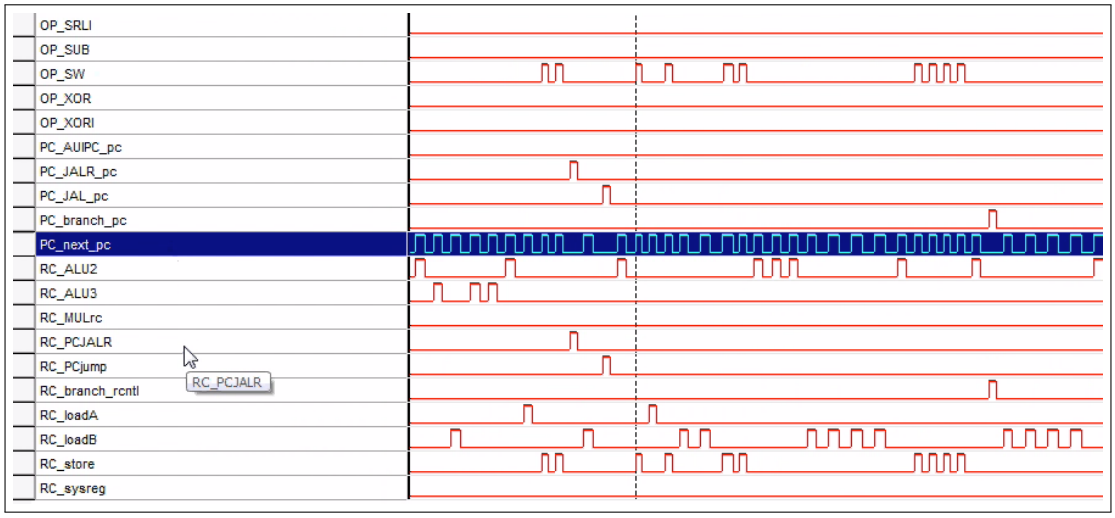

Figure 138: SingalTap FPGA Internal Signal Monitoring
Instruction Decoder Control Outputs

appear quite uniform. If the execution gate count is different then the execution time will be different, as can be seen in the waveform. As mentioned earlier, because this implementation still uses synchronous memories for program and data (i.e, Block RAMs within the FPGA), the minimum cycle of each instruction also longer than the Memory Access Time.

5.3 Asynchronous RISC-V CPU Architectural Optimization

This section presents a number of architectural-level optimizations performed during the development of the asynchronous CPU. It is explained here why the optimization strategies for asynchronous CPU systems are quite different to conventional design. Further, using

statistical results derived from the RISC-V benchmark tests, it is shown how the CPU design was optimized at the architectural level.

5.3.1 CPU Architectural Optimization

In the previous chapter, the CPU component design and their component level optimization techniques were described. In the case of conventional synchronous design, RTL designers can implement CPU components directly from Verilog expressions with the help of synthesis tools such as DesignWare from Synopsys and ChipWare from Cadence. Synthesis tools such as Synopsys Design-Compiler and Cadence Genus have their own libraries and IP blocks from the tool vendors which are pre-designed and well optimized for the particular process. DesignWare and ChipWare are parameterizable libraries that allow the synthesis tools to automatically select the optimum arithmetic components based on the designer's performance, area and power targets. Thus, the RTL designer need not worry about the details of the CPU components as long as the output net-list meets their requirements. Unfortunately, no similar tools or libraries exist to date for asynchronous design. Thus it has been necessary design all of the specific CPU arithmetic components at industry standard level. This was the subject of Chapter 4.

Once the suitable CPU components have been derived, the CPU architecture-level optimizations can proceed. Compared to the synchronous case, a different strategy is necessary for the optimization of an asynchronous CPU. In the synchronous case, we are always considering the worst-case delay when attempting to increase overall performance of the CPU. On the other hand, asynchronous timing is based on average-case performance. Therefore, the dynamic (run-time) operation of the machine is more important than its static analysis. This requires detailed statistical analysis (i.e., *profiling*) of the run-time flow, in order to provide information for the optimization steps. Conventional synchronous synthesis tools do not support this type of analysis.

For example, in the Dhrystone benchmark test, the multiplication instruction makes up just 0.5% of the dynamic count but the addition instruction represents more than 25%. In synchronous design, the Adder delay is normally much shorter than the multiplier delay, which tends to push the design in the direction of using a smaller, slow adder but a larger and faster multiplier block to meet the timing requirements. Even then the multiplier would

typically require multiple pipeline stages to meet the performance demands. In contrast, asynchronous CPU design may use a fast adder and the smallest multiplier to improve average-case performance while reducing cell area, because the dynamic frequency of the addition instruction is 50x more than that of multiplication.

Of course, the dynamic instruction results will vary widely from one application to another which would imply that we would have to optimize for a specific case, or to provide customized IP blocks based on the application. However, in many cases we can expect the instruction usage to be relatively stable across a range of applications in a similar domain, especially when the application has a special purpose such as embedded system or IoT device. In this case the dynamic instruction estimates will still be valid.

5.3.2 Prioritized NCL-based RISC-V Optimization

In synchronous design, the timing tools forming part of the Synthesis, Place and Route, and Static Timing Analysis tools will always try to minimize the worst case timing delay between flip-flop outputs and the next flip-flop inputs based on the input timing constraints. With the exception of the critical path, most of the timing paths would be relatively relaxed because they have enough margin to meet the timing constraints. In this case, the relaxed timing means that the tool does not need to increase the driving strength of the cells, the cells do not need to be placed close to one another and the interconnect routing does not need to necessarily follow the shortest path. Therefore the timing calculation tool always has the priority from the worst case path to the best case path based on designer's input timing constraints as specified, for example, in the SDC (Synopsys Design Constraint) file. As a result, a designer must always consider the possible worst case timing path even at the RTL design stage and try to reduce the worst case delay as much as they can.

Asynchronous design is a very different story. The performance of an asynchronous CPU is determined by its average delay time per cycle. Some instruction cycle times would be relatively longer than average and some are the same or shorter. The critical timing path in this case (i.e., the path that will determine the overall machine performance) would be formed by the most frequently used instructions and their execution logic. On the other hand, the least used instructions and their timing paths would be less important for the optimization and their timing can be relaxed regardless their relative delay time. Thus, an asynchronous timing tool will have different priorities than the synchronous case, and this needs to be considered when

developing the RTL. Unfortunately, timing and optimization tools that support this type of dynamic critical path approach are not yet available (and are beyond the scope of this thesis). As a result, this section examines how the architecture and RTL designers can approach the asynchronous timing optimization task at the design stage.

5.3.3 RISC-V Benchmark Instruction Statistics

The RISC-V CPU dynamic instruction statistics have been generated using benchmarks drawn from the UC-Berkeley RISC-V benchmark test repository [137] which contains a total of 12 test-case: dhrystone, median, mm, mt-matmul, mt-vvadd, multiply, pmp, qsort, rsort, spmv, towers and vvadd. Just as for the ISA verification tests in Section 5.2, these benchmark tests use the RISC-V compiler tool-chain that generates compiled ELF (Executable and Linkable Format) files. These were converted to .HEX format using the RISC-V elf2hex tool before Verilog simulation and transfer to FPGA. The benchmarks have been simulated using the Spike [138] RISC-V ISA Simulator and the instructions were counted using a custom Python script. The statistics were created from ten of the twelve test cases: dhrystone, median, mt-matmul, mt-vvadd, multiply, qsort, rsort, spmv, towers and vvadd[4].

The following is a brief description of each benchmark [137]:

Dhrystone: Dhrystone is a synthetic computing benchmark program that is intended to measure the effect of integer programming. Dhrystone is seen as generally representative of CPU performance. Unlike MIPS (Millions of Instructions per Second), the Dhrystone score counts only the number of iterations per second [139]. Because the generated instruction count is compiler dependent, it is a better measure than MIPS when comparing different CPU instruction architectures.

Median: Median benchmark performs an 1D three element median filter. Because this benchmark test encompasses a lot of memory access instructions such as load and store, it provides a good means to test memory access performance.

Mt-matmul: Multi-threaded Matrix Multiply benchmark. This benchmark multiplies two 2D arrays together and writes the results to a third vector. This test uses a lot of multiply instructions.

[4]mm and pmp have been omitted here as the tests in the RISC-V repository are not working correctly, as yet. This is a repository issue beyond the control of this thesis.

Mt-vvadd: The Multi-threaded Vector-vector addition benchmark is a vector addition program that heavily uses the ADDI instruction.

Multiply: Multiply filter benchmark. This benchmark tests the software multiply implementation. It does not directly use the Multiply instruction, to avoid the need for a hardware multiplier unit.

Qsort: The Quicksort benchmark uses the quicksort algorithm to sort an array of integers. The implementation is largely adapted from *Numerical Recipes for C*. This test uses a lot of ADDI, branch and load/store instructions.

Rsort: Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value [140].

Spmv: Sparse matrix-vector multiplication

Towers: Towers of Hanoi benchmark. Towers of Hanoi is a classic puzzle [141]. This benchmark test uses a lot of load/store instructions with ADDI.

Vvadd: Vector-vector add benchmark

Table 27 shows the RISC-V RV32IM instruction statistics derived from the ten benchmark tests described above. The benchmark tests used a total of 38 out of the 45 instructions that have been implemented on this core. The RISC-V compiler generates a small number of *pseudo-Instructions* that are not actual RISC-V instructions but are allocated by the compiler to real instructions and which were counted in the results. The lower part of the table has 11 instruction groups. The details of the groupings will be explained on the next sub-section.

Clearly, these benchmark tests do not represent all the possible application cases but they are still good examples to show how the asynchronous CPU designs are optimized using their run-time profile statistics.

	Real Instruction	Pseudo Instruction	dhrystone	%	median	%	mt-matmul	%	mt-vvadd	%	multiply	%	qsort	%	rsort	%	spmv	%	towers	%	vvadd	%	Average%
1	add		5219	2.45	61	0.41	8969	34.11	5	1.92	1035	2.31	2821	1.23	51317	13.87	140313	8.52	57	0.39	657	6.59	7.18
2	addi	li, mv	55079	25.88	2761	18.45	2776	10.56	96	36.78	8384	18.73	76484	33.26	53032	14.33	274713	16.67	5399	36.70	2812	28.21	23.96
3	and			0.00		0.00		0.00		0.00		0.00		0.00		0.00	114504	6.95		0.00		0.00	0.70
4	andi		4016	1.89	10	0.07	260	0.99	4	1.53	6410	14.32	560	0.24	32794	8.86	54094	3.28	11	0.07	10	0.10	3.14
5	auipc		7624	3.58	29	0.19	8	0.03	7	2.68	30	0.07	33	0.01	36	0.01	38	0.00	48	0.33	30	0.30	0.72
6	beq	beqz	28063	13.19	759	5.07	259	0.99	2	0.77	6594	14.73	3482	1.51	761	0.21	42552	2.58	561	3.81	499	5.01	4.79
7	bge	bgez	1031	0.48	1112	7.43		0.00		0.00	24	0.05	6333	2.75	24	0.01	33800	2.05	28	0.19	24	0.24	1.32
8	bgeu		696	0.33	83	0.55	19	0.07	3	1.15	105	0.23	677	0.29	4239	1.15	19279	1.17	83	0.56	83	0.83	0.63
9	blt	bltz	520	0.24	1073	7.17	1	0.00	1	0.38	11	0.02	37587	16.35	13	0.00	12487	0.76	9	0.06	9	0.09	2.51
10	bltu		1025	0.48	21	0.14	33	0.13	33	12.64	21	0.05	7136	3.10	4133	1.12	4819	0.29	21	0.14	21	0.21	1.83
11	bne	bnez	2199	1.03	1475	9.86	784	2.98		0.00	6835	15.27	6235	2.71	9299	2.51	32683	1.98	116	0.79	1129	11.33	4.85
12	csrrs	csrr, csrs	9	0.00	7	0.05	7	0.03	5	1.92	7	0.02	7	0.00	7	0.00	7	0.00	7	0.05	7	0.07	0.21
13	csrrw	csrw	1	0.00	1	0.01	1	0.00	1	0.38	1	0.00	1	0.00	1	0.00	1	0.00	1	0.01	1	0.01	0.04
14	div		502	0.24		0.00		0.00		0.00		0.00		0.00		0.00		0.00		0.00		0.00	0.02
15	divu		68	0.03	28	0.19	1	0.00		0.00	36	0.08	44	0.02	44	0.01	44	0.00	28	0.19	28	0.28	0.08
16	jal	j	8663	4.07	577	3.86	8	0.03	7	2.68	261	0.58	7188	3.13	79	0.02	24355	1.48	374	2.54	53	0.53	1.89
17	jalr	jr	140	0.07	35	0.23	1	0.00	1	0.38	37	0.08	39	0.02	39	0.01	49	0.00	35	0.24	35	0.35	0.14
18	lbu		24205	11.38	79	0.53		0.00		0.00	85	0.19	91	0.04	91	0.02	91	0.01	79	0.54	79	0.79	1.35
19	lhu		502	0.24		0.00		0.00		0.00		0.00		0.00		0.00		0.00		0.00		0.00	0.02
20	lui		575	0.27	31	0.21	2	0.01	2	0.77	39	0.09	50	0.02	48	0.01	81245	4.93	31	0.21	31	0.31	0.68
21	lw		32463	15.26	4756	31.78	8199	31.18	2	0.77	1041	2.33	56327	24.50	77337	20.90	118535	7.19	3759	25.55	2852	28.61	18.81
22	mul		1069	0.50	28	0.19	4099	15.59		0.00	36	0.08	44	0.02	44	0.01	76652	4.65	28	0.19	28	0.28	2.15
23	or		3327	1.56	770	5.15	5	0.02	5	1.92	213	0.48	235	0.10	777	0.21	124051	7.53	190	1.29	510	5.12	2.34
24	ori		1	0.00	1	0.01		0.00		0.00	1	0.00	1	0.00	1	0.00	1	0.00	1	0.01	1	0.01	0.00
25	remu		68	0.03	28	0.19		0.00		0.00	36	0.08	44	0.02	44	0.01	44	0.00	28	0.19	28	0.28	0.08
26	sb		1635	0.77	32	0.21		0.00		0.00	34	0.08	36	0.02	36	0.01	36	0.00	32	0.22	32	0.32	0.16
27	sh		502	0.24		0.00		0.00		0.00		0.00		0.00		0.00		0.00		0.00		0.00	0.02
28	sll		102	0.05	42	0.28		0.00		0.00	54	0.12	66	0.03	66	0.02	6206	0.38	43	0.29	42	0.42	0.16
29	slli		1689	0.79	84	0.56	6	0.02	5	1.92	6502	14.53	672	0.29	49276	13.32	154968	9.41	82	0.56	82	0.82	4.22
30	sltiu	seqz	1500	0.70		0.00	1	0.00	1	0.38		0.00		0.00		0.00	250	0.02		0.00		0.00	0.11
31	sltu	snez	15	0.01	6	0.04		0.00		0.00	8	0.02	10	0.00	10	0.00	40392	2.45	7	0.05	6	0.06	0.26
32	srai		6	0.00	2	0.01		0.00		0.00	6402	14.30	2742	1.19	2	0.00	2	0.00	2	0.01	2	0.02	1.55
33	srl		87	0.04	36	0.24		0.00		0.00	46	0.10	56	0.02	32824	8.87	6196	0.38	36	0.24	36	0.36	1.03
34	srli		136	0.06	56	0.37	256	0.97		0.00	72	0.16	636	0.28	88	0.02	173874	10.55	56	0.38	56	0.56	1.34
35	sub		1098	0.52	43	0.29	258	0.98	2	0.77	53	0.12	3353	1.46	71	0.02	12329	0.75	45	0.31	43	0.43	0.56
36	sw		28949	13.60	939	6.27	340	1.29	79	30.27	350	0.78	16963	7.38	53406	14.44	86908	5.28	3514	23.89	743	7.45	11.07
37	xor			0.00		0.00		0.00		0.00		0.00		0.00		0.00	4798	0.29		0.00		0.00	0.03
38	xori	not		0.00		0.00		0.00		0.00		0.00		0.00	16	0.00	7170	0.44		0.00		0.00	0.04
	Total		212784	100.00	14965	100.00	26293	100.00	261	100.00	44763	100.00	229953	100.00	369955	100.00	1647486	100.00	14711	100.00	9969	100.00	99.96
1	ALU Immediate Group		62427	29.34	2914	19.47	3299	12.55	106	40.61	27771	62.04	81095	35.27	135209	36.55	665072	40.37	5551	37.73	2963	29.72	34.36
2	ALU Group		9848	4.63	958	6.40	9232	35.11	12	4.60	1409	3.15	6541	2.84	85065	22.99	448789	27.24	378	2.57	1294	12.98	12.25
3	CSR Group		10	0.00	8	0.05	8	0.03	6	2.30	8	0.02	8	0.00	8	0.00	8	0.00	8	0.05	8	0.08	0.25
4	Branch Group		33534	15.76	4523	30.22	1096	4.17	39	14.94	13590	30.36	61450	26.72	18469	4.99	145620	8.84	818	5.56	1765	17.70	15.93
5	Load Immediate Group		575	0.27	31	0.21	2	0.01	2	0.77	39	0.09	50	0.02	48	0.01	81245	4.93	31	0.21	31	0.31	0.68
6	Load Group		57170	26.87	4835	32.31	8199	31.18	2	0.77	1126	2.52	56418	24.53	77428	20.93	118626	7.20	3838	26.09	2931	29.40	20.18
7	Store Group		31086	14.61	971	6.49	340	1.29	79	30.27	384	0.86	16999	7.39	53442	14.45	86944	5.28	3546	24.10	775	7.77	11.25
8	Jump Group A		140	0.07	35	0.23	1	0.00	1	0.38	37	0.08	39	0.02	39	0.01	49	0.00	35	0.24	35	0.35	0.14
9	Jump Group B		16287	7.65	606	4.05	16	0.06	14	5.36	291	0.65	7221	3.14	115	0.03	24393	1.48	422	2.87	83	0.83	2.61
10	MUL Group		1069	0.50	28	0.19	4099	15.59	0	0.00	36	0.08	44	0.02	44	0.01	76652	4.65	28	0.19	28	0.28	2.15
11	DIV Group		638	0.30	56	0.37	1	0.00	0	0.00	72	0.16	88	0.04	88	0.02	88	0.01	56	0.38	56	0.56	0.18
			212784	100.00	14965	100.00	26293	100.00	261	100.00	44763	100.00	229953	100.00	369955	100.00	1647486	100.00	14711	100.00	9969	100.00	100.00

Table 27: RISC-V RV32IM Instruction Statistics

5.3.4 Classification of RISC-V Instructions

As we introduced in Table 23, RISC-V is a simplified and well organized ISA and the instructions fall into clearly defined groups. The RISC-V Base Integer ISA has six instruction types (R/I/S/B/U/J) and all the instructions can be allocated within those groups. Table 28 shows the RISC-V RV32IM instruction grouping and their applications, extracted from the benchmark test simulation. This table includes all of the 45 instructions that were implemented. By rearranging these groups as in Figure 125, it can be seen that the Redback RISC has four execution groups, ALU, Multiplier, Load and Store Unit and Branch Unit.

On Table 28, Group 1 is the ALU-Immediate Group and includes all of the instructions related to the ALU and I-type instructions. Group-2 is the ALU Group, the same as the ALU-Immediate Group, but here the instructions are R-type. Group-3 is the Branch Group and Group-4 is the Load Immediate Group. The LUI instruction is the only U-type instruction between the Load instructions. Group-5 represents the Load Group and these are I-type instructions. Group-6 is the Store Group and are also I-type instructions. Group-7 comprises JALR instruction and are I-type instructions while Group-8 has JAL and AUIPC and they are J-type and U-type instructions. The only difference is the immediate bit allocation but rest of them are the same. Group-9 is the MULT Group. While the Multiplication and Division instructions are still R-type instructions, because these have separated execution block, it was transferred into a separate group. The bottom of Table 27 has 11 groups but in Table 28 the MUL and DIV groups have been merged and the CSR (Control and Status Registers) group is not implemented and treated as NOP (No Operation) in our design.

5.3.5 Statistical Priority Approach to Machine Optimization

As outlined above, the optimization of the critical timing paths in an asynchronous machine needs to be informed by the most frequently used instructions. Focusing on the most used execution logic, while relaxing the relative timing of the least used instructions is most likely to result in optimum overall machine performance.

The instruction decoder and execution modules such as the ALU and multiplier are the most obvious choices as to where to apply this statistical priority (or profiling) approach. However, in the case of the instruction decoder, if it is not possible to alter the actual instruction encoding, there will be limited opportunities to optimise the logic in critical path of the

Group No.	Group Name	Instruction	Opcode [6:0]	Function3 [14:12]	Function7 [31:25]	Instuction Meaning	Average Instruction Count %	Group %
1	ALU Immediate Group	ADDI	001_0011	000		Add Immediate	23.96	34.36
		SLTI		010		Set Less Than Immediate	0.00	
		SLTIU		011		Set Less Than Immediate Unsigned	0.11	
		XORI		100		XOR Immediate	0.04	
		ORI		110		OR Immediate	0.00	
		ANDI		111		AND Immediate	3.14	
		SLLI		001	000_0000	Logical Left Shift Immediate	4.22	
		SRLI		101	000_0000	Logical Right Shift Immediate	1.34	
		SRAI		101	010_0000	Arithmetic Right Shift Immediate	1.55	
2	ALU Group	ADD	011_0011	000	000_0000	ADD	7.18	12.26
		SUB		000	010_0000	SUB	0.56	
		SLL		001	000_0000	Logical Left Shift	0.16	
		SLT		010	000_0000	Signed Compare	0.00	
		SLTU		011	000_0000	Unsigned Compare	0.26	
		XOR		100	000_0000	XOR	0.03	
		SRL		101	000_0000	Logical Right Shift	1.03	
		SRA		101	010_0000	Arithmetic Right Shift	0.00	
		OR		110	000_0000	OR	2.34	
		AND		111	000_0000	AND	0.70	
3	Branch Group	BEQ	110_0011	000		Branch Equal	4.79	15.93
		BNE		001		Branch Unequal	4.85	
		BLT		100		Branch Less Than	2.51	
		BGE		101		Branch Greater than or Equal	1.32	
		BLTU		110		Branch Less Than Unsigned	1.83	
		BGEU		111		Branch Greater than or Equal Unsigned	0.63	
4	Load Immediate Group	LUI	011_0111			Load Upper Immediate	0.68	0.68
5	Load Group	LB	000_0011	000		Load Byte Sign Extends	0.00	20.18
		LH		001		Load Half Sign Extends	0.00	
		LW		010		Load Word	18.81	
		LBU		100		Load Byte Zero Extends	1.35	
		LHU		101		Load Half Zero Extends	0.02	
6	Store Group	SB	010_0011	000		Store Byte	0.16	11.25
		SH		001		Store Half	0.02	
		SW		010		Store Word	11.07	
7	Jump Group A	JALR	110_0111	000		Jump And Link Register	0.14	0.14
8	Jump Group B	JAL	110_1111			Jump And Link	1.89	2.61
		AUIPC	001_0111			Add Upper Immediate to PC	0.72	
9	MULT Group	MUL	011_0011	000	000_0001	MUL return Lower 32bits	2.15	2.25
		MULH		001	000_0001	MUL return Upper 32bits, Signed x Signed	0.00	
		MULHSU		010	000_0001	MUL return Upper 32bits, Signed x Unsigned	0.00	
		MULHU		011	000_0001	MUL return Upper 32bits, Unsigned x Unsigned	0.00	
		DIV		100	000_0001	Signed Division	0.02	
		DIVU		101	000_0001	Unsigned Division	0.08	
		REM		110	000_0001	Signed Remainder	0.00	
		REMU		111	000_0001	Unsigned Remainder	0.08	

Table 28: RISC-V RV32IM Instruction Grouping and Their Usage Table

decoder for specific instructions, even though we might want to assign a higher priority to those instruction. The RISC-V ISA has a well organized instruction set and even though it supports a number of user-define instruction opportunities at the ISA-Level and also the compiler level, there is not much room to add higher priority instructions unless we totally change the ISA itself. It may be possible to perform a binary re-coding of the opcodes that could result in shorter critical paths in the decoder, but this is beyond the scope of this current work. A binary translation scheme of this sort will be left for future work. Further, it has been found that the Instruction Decoder module is not usually on the critical path therefore it will be more helpful to optimize the execution modules, which was the main objective of the work in Chapter 4.

5.4 RISC-V Performance Comparisons

This section will compare the Redback RISC with some existing synchronous RISC-V designs already available in the wider RISC-V community. A more complete list of the known RISC-V cores and SoCs that have been developed and released can be found at the RISC-V Foundation web-site [142], As the RISC-V ISA is an open instruction architecture, many of these RISC-V designs are also in the public domain. Because our Redback RISC core has a 32-Bit architecture and supports RV32IM instruction sets, similar architectures have been selected for comparison purposes. While these are not identical, most of their functions are similar and also the cores are suitable to synthesise and run the riscv-tests benchmarks [137] with the same test images compiled using the 32-bit RISC-V compiler environment. Therefore, the comparison results will be relatively fair. The two cores selected are the PicoRV32 [143] and Rocket [144].

PicoRV32 was designed by Clifford Wolf and initially released in May 2015. PicoRV32 was optimized for area and also aimed to achieve a high f_{max}. The high clock speed target has meant that the average CPI (Cycles per Instruction) of PicoRV32 is approximately four [143], which means it needs multiple clocks to execute one instruction. The PicoRV32 Dhrystone benchmark test results are 0.516 DMIPS/MHz and 908 Dhrystones/Sec/MHz based on published results [143]. The PicoRV32 is used as the core within PicoSoC and is also used for many FPGA and ASIC projects as a standard 32-bit RISC-V core.

The Rocket core was designed by the UC-Berkeley group [145], initially to evaluate the RISC-V ISA. It was designed using the Chisel open-source hardware construction language embedded in Scala [146], all developed at UC-Berkeley. The Rocket core was initially uploaded in October 2011 by Rimas Avizienis and is managed via the "Rocketchip Generator'' repository. After that initial upload, the repository has been updated by numerous engineers and is now managed by the "Chips Alliance'' [147]. Rocket is the main CPU core employed by SiFive Co. Ltd [148] and has been fabricated many times for their own production chips as well as for their IP customers. Chisel is an easily parameterizable language and with simple configuration changes, it can generate CPU architectures with many different options. The Chisel compiler generates a Verilog RTL file that can be used for either FPGA or ASIC implementations (or both). Memories are not included in the Chisel output file but the Verilog RTL does include memory interfaces. The Configuration options included in the Rocketchip Generator [145] are shown in Table 29. The *TinyConfig* mode has been selected for this comparison because it is the

smallest configuration and also includes the 32-Bit Rocket Core. It will be referred to below as *Rocket-Tiny* to indicate this combination.

BaseConfig, DefaultConfig	Used when no other configurations are specified
DefaultBufferlessConfig, DefaultSmallConfig	Removes Floating Point Unit and has smaller caches
DefaultRV32Config, RoccExampleConfig DualCoreConfig, DefaultFPGAConfig	Used in specific cases
TinyConfig	Smallest configuration and also includes the 32-Bit Rocket Core
User Options	User definable configurations

Table 29: Rocketchip Generator Configuration Options

5.4.1 Synchronous RISC-V CPU design

Both PicoRV32 and Rocket-Tiny cores are implemented using the standard synchronous ASIC tool flow (Figure 139). A commercial 28nm Standard Cell Library has been used for this standard synchronous implementation. As mentioned above, the RTL descriptions were simulated using the riscv-tests benchmark images (.hex files) using the Synopsys VCS simulation tools.

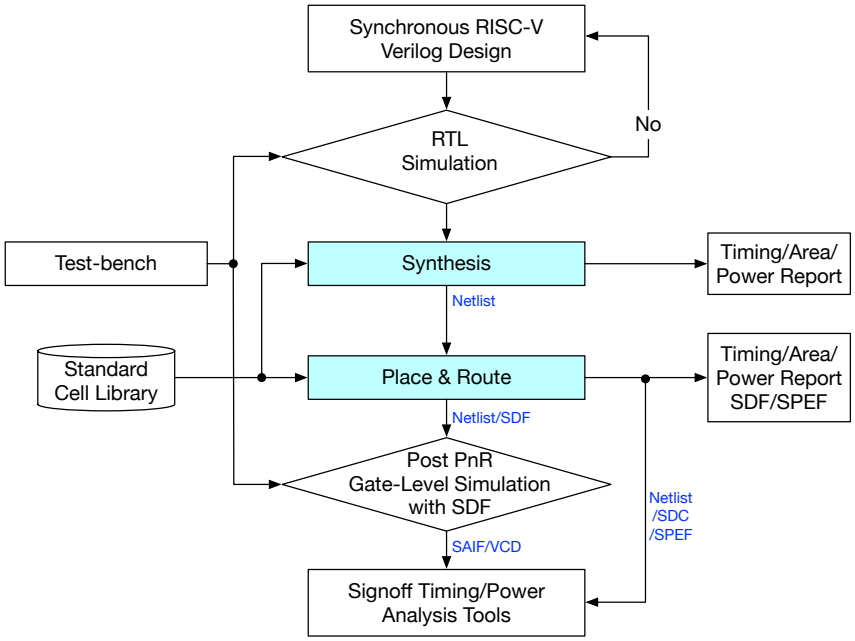

Figure 139: Synchronous Design Flow

The RTL codes were first synthesized using the Synopsys Design-Compiler synthesis tools and then placed and routed (PnR) using the Synopsys IC-Compiler PnR tools. The

placed and routed designs were simulated using the same test-bench files but now with the SDF (Standard Delay Format) files to produce switching activity files such as SAIF (Switching Activity Interface Format) and VCD (Value Change Dump) files for the sign-off timing and power analysis tools. Finally, the timing and power figures for the placed and routed net-list were measured using Synopsys PrimeTime and PrimeTime-PX tools for timing and power respectively. Both timing and power tools need post PnR net-list file, SDC (Synopsys Design Constraints), and SPEF (Standard Parasitic Exchange Format) files with the switching activity files (SAIF/VCD) generated by the VCS simulation tools. This is a quite standard design flow to achieve accurate PPA results, except for the detailed sign-off steps more usually reserved for chip tape-out. Extraction tools can be used to obtain more accurate results that take parasitic components into account, but it is likely that the SPEF file generated by the IC-Compiler PnR tool will be sufficiently accurate for these experiments, particularly as they are more concerned with measuring the *ratio* of the performance between the various CPUs.

5.4.2 Asynchronous RISC-V CPU Design using UNCLE

As introduced in Chapter 3, Section 3.1, UNCLE is currently the major design tool for Null Convention Logic technology development. Figure 140 shows the NCL based RISC-V CPU core design flow using UNCLE, in which a clocked Boolean synchronous design is converted to its corresponding NCL topology. Firstly as was done previously for the synchronous RISC-V design implementation, the RTL simulation of the synchronous PicoRV32 and Rocket-Tiny designs were tested using Synopsys VCS. Once the RTL simulation had passed, the RTL was converted to NCL using the UNCLE tools. UNCLE internally uses synchronous synthesis tools to generate the synchronous net-list before it converts to the NCL net-list. The scripts supports two different synthesis tools - Synopsys Design-Compiler and Cadence RTL-Compiler (now upgraded to Genus). This work has used Design-Compiler. In contrast to the synchronous design flow of Figure 139, UNCLE uses a special cell library defined internally in the tool to perform the conversion to NCL. The library has only very limited set of cells and these will be 1:1 converted to NCL cells based on the internal mapping tables built into the tool.

UNCLE is not a commercial tool and its regression tests mostly use block level synchronous designs. However, the comparison tests in this thesis needed to convert full 32-bit CPU cores, which was difficult to achieve using a simple tool like UNCLE. As a result, UNCLE failed quite often, mostly crashing i.e., exiting unexpectedly in the middle of its flow. As a

result, although we actually tested most of the RISC-V CPU cores [142], only few cores were successfully translated into a final NCL net-list. UNCLE also has significant RTL code limitations which are explained in their documentation [15]. An UNCLE flow wrapper was developed for this work that uses Python scripts to automatically control the flow also to produce an area report after conversion (Appendix-A). It was necessary to add a couple of work-arounds to compensate for issues arising from these complex designs.

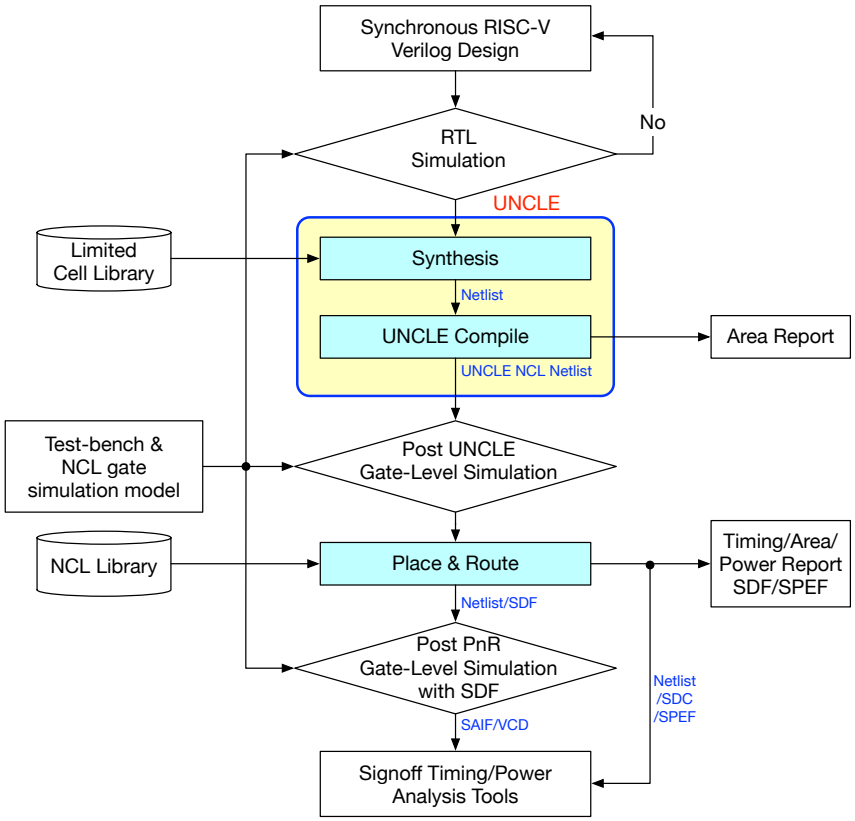

Figure 140: UNCLE Design Flow

UNCLE includes functional simulation as one of its outputs but this was not used in this flow. Instead of that we have just used the UNCLE output net-list and simulated the net-list independently using the same test-benches used for the input RTL simulation. For this simulation, it was necessary to include an NCL gate simulation model that has fixed delay values for each of the different NCL cells and their delay values are extracted from the cell Spice simulation. Once the UNCLE's output gate-level net-list was simulated, the next step was the Place and Route using a 28nm NCL cell library built with the same technology used for the synchronous RISC-V. After the PnR, the final step is same as synchronous design, using the same test-bench. The PnR net-list is simulated with the SDF delay values and the results will be analyzed using the Sign-off Timing and Power analysis tools, in our case PrimeTime

and PrimeTime-PX.

5.4.3 Asynchronous RISC-V CPU design using NELL - Redback RISC

The Redback RISC NCL CPU core has been designed using the NELL language developed by Wave Computing [4] [5]. As was addressed in Chapter 3, Section 3.1, NELL provides language, compilation and simulation functions. In this design, the NELL language and compiler have been used but not the simulator, because the NELL simulator is harder to use for debugging purposes than standard Verilog simulators (such as VCS of Synopsys, NC-Sim of Cadence, QuestaSim of Mentor Graphics) and also has limited functionality at present. Because the same test-bench as was used for both the synchronous and UNCLE RISC-V cores is being applied here, it was considered to be much better to use the standard simulation tools. The rest of flow is the same as that used in the UNCLE -based case. The net-list is placed and routed using standard PnR tools (Synopsys IC-Compiler in this case) and the post-PnR net-list simulated with SDF delay values. Finally, the results are transferred to the sign-off timing and power analysis tools. Figure 141 explains the flow details.

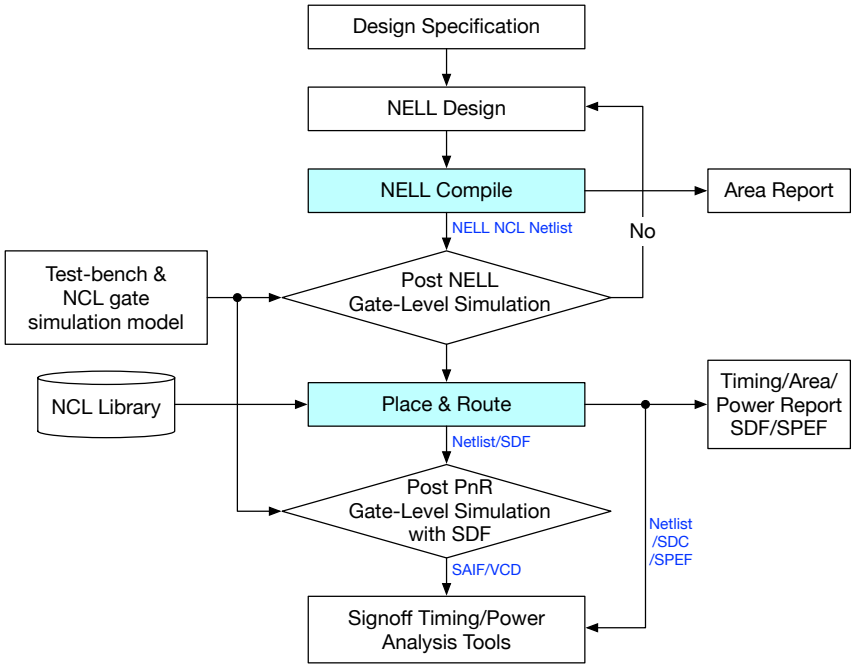

Figure 141: NELL Design Flow

In these comparison experiments, the same program and data memories have been used across all of the designs, resulting in identical memory access times in all cases. Usually,

[5] The RMIT research group was involved in the design of NELL and used the examples of the Redback RISC design plus all the CPU sub-modules to stress-test the tool. We reported many tool issues and also suggested many new functions that helped its development

the instruction and data cache interfaces and their controller performance are amongst the more important determinants of CPU performance but in these test cases, the CPU cores do not include the cache controllers. Therefore the memory interface timing will not affect the overall CPU performance meaning that these experiments are purely CPU core performance tests.

5.5 RISC-V Design Comparison and Benchmark Tests

This section compares Redback RISC performance to a small number of approximately equivalent synchronous RISC-V CPU cores. The comparison has been based on the Dhrystone benchmark test as this benchmark is widely used for 32-bit CPU core comparison purposes.

5.5.1 Redback RISC Implementation

As explained in the previous section, the Redback RISC core is designed using the NELL programming language and the NELL compiler generates the NCL net-list for the subsequent Place and Route step. Figure 142 shows the floor-plan of the machine after place and route. This implementation uses a hierarchical net-list to show the approximate hierarchical area results. In the final implementation, a fully flattened net-list was used. The 32-entry, 32-bit RISC-V Register File block occupied more than 1/3 of the total area and the multiplier block also takes up more than one quarter of the area. The place and route is constrained to 70% utilization with an aspect-ratio of one.

5.5.2 RISC-V CPU Comparisons

These comparison tests have used the same 28nm technology, with a supply of 0.9V, 25°C in this case. As explained in section 5.4, three alternative tool flows have been used. These tool flows have used Synopsys VCS simulation tools for the RTL and Gate-Level simulation and Design-Compiler for the RTL Synthesis. Synopsys IC-Compiler tools are used for the Auto Place and Route and Synopsys PrimeTime and PrimeTime-PX tools are used for the Sign-off timing and power analysis.

The comparison uses the PicoRV32 and the Rocket core with the TinyConfig option. The Rocket core was initially designed by UC-Berkeley using Chisel [149] hardware construction language and the design is converted to the RTL Verilog using the Chisel compiler. Rocket TinyConfig option generates a 32-bit RISC-V core and this is the smallest option available from the Rocket-Chip Generator [144].

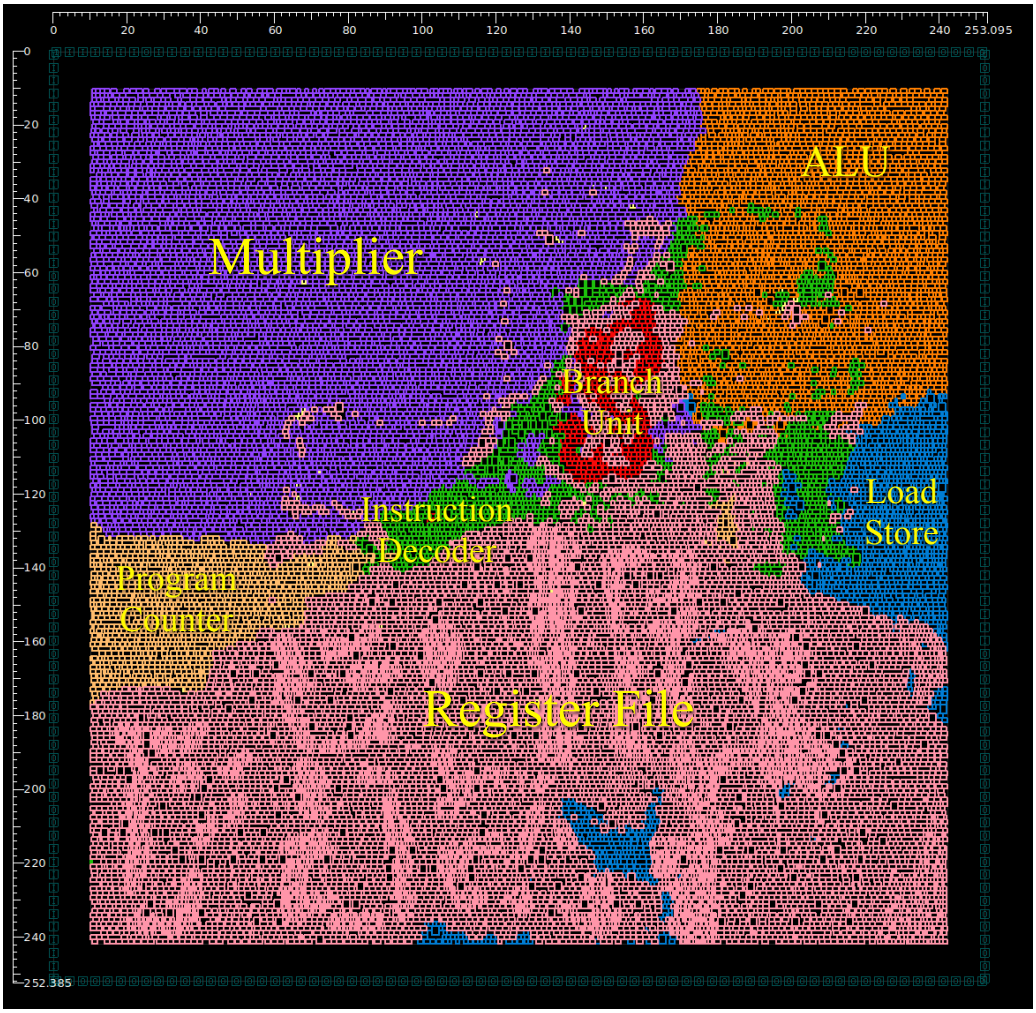

Figure 142: Redback RISC Floor-plan (unit: μm)

Table 30 and Figure 143 show the comparison results and includes five different cases: the PicoRV32, the PicoRV32 converted using UNCLE, the Rocket-Tiny and its UNCLE translation, and the Redback. PicoRV32 [143] is a synchronous design from the github and it is directly implemented and tested using a conventional synchronous tool flow (Figure 139). PicoRV32 UNCLE is the design converted from the PicoRV32 to the UNCLE net-list using the tool flow (Figure 140). The synchronous Rocket-Tiny was also downloaded from the github [144] and Rocket-Tiny UNCLE is the design after conversion from the Rocket-Tiny to the NCL net-list using the UNCLE tools. Finally, Redback is the NCL based RISC-V design.

We expected a clock speed of about 3GHz for PicoRV32 because the core is deeply pipelined and high performance is one of its stated objectives. However, in this 28nm technology, a maximum speed of only 1.5GHz could be achieved without worst-case negative slack. The core had slightly worst-case negative slack even with 2GHz clock constraints, therefore

the gate-level simulation on the 2GHz clock failed. Rocket-Tiny core achieved 1GHz without worst-case negative slack after place and route, but it was necessary to use the *High Effort* option to get these results.

For the Redback core, the Dhrystone benchmark performance results were used to determine the executed instruction count and simulation time. It has 202,931 instructions executed in 546μS. The performance of Redback is the averaged Million Instructions per Second (MIPS) figure rather than the cycle frequency because the NCL based CPU execution time is quite variable and very dependant on the input operands.

As described on the PicoRV32 web-site [143], PicoRV32's average Cycles per Instruction (CPI) is approximately four so, even though its clock speed is 1.5GHz, the overall Dhrystone execution time is much longer than the 1GHz Rocket-Tiny design. The second and fourth columns are the UNCLE net-list implementation comparison results. These show that the UNCLE net-list is approximately 8$\times$ larger than the equivalent synchronous design. Table 30 and Figure 144 show the core area comparisons. Again, the place and route is constrained to 70% utilization with an aspect-ratio of 1. The real size of the image is roughly 30% larger than the actual cell area in Table 30 because of the 70% cell utilization.

5.5.3 RISC-V Benchmark test

The Dhrystone program was originally designed by Reinhold Weicker [139] in 1984. It is used to measure the integer processing performance of computing systems and therefore contains no floating point operations. Dhrystone may represent a processor more accurately than MIPS (Million Instructions per Second) because different types of machine architectures (e.g., CISC vs. RISC) use completely different compiler structures and thus their compiled instruction count for the same program will be quite different. Thus, the Dhrystone score counts only the number of program iteration completions per second, allowing individual machines to perform this calculation in a machine-specific way [139]. A common Dhrystone metric is the DMIPS (Dhrystone MIPS), which is the Dhrystone score normalized by dividing by 1757, representing the number of Dhrystone per second obtained on the VAX11/780 machine, a 1 MIPS machine. Because we are using the same RISC-V instructions and the same RISC-V compiler, MIPS still represents a useful comparison. On the other hand, the DMIPS figures can be used to compare with other ISA machines such as ARM. An alternate representation, DMIPS/MHz, divides the DMIPS result by the CPU clock frequency and therefore allows easier comparison

between CPUs running at different clock rates. However, because we are comparing the same architecture with the same conditions, the DMIPS comparison will be more useful than DMIPS/MHz. Further, the DMIPS/MHz figure has different meaning for asynchronous processors because, by definition, these lack a clock signal and each instruction completes at an arbitrary time when the final acknowledgement signal completes before the next instruction fetch.
The equations for the DMIPS calculation are given as:

$$DMIPS = Number\ of\ Dhrystone\ Cycles/Execution\ Time \tag{5.1}$$

$$DMIPS/MHz = 10^6/(1757 * Number\ of\ processor\ clock\ cycles\ per\ Dhrystone\ loop) \tag{5.2}$$

In this case, the simulation is running 500 Dhrystone cycles. From (5.1) the Redback RISC core runs at 520 DMIPS (Dhrystone V2.1) and 1.404 DMIPS/MHz (from 5.2). Table 30 shows the Dhrystone DMIPS vs. DMIPS/MHz comparison results. Because the same RISC-V compiler has been used, along with the same benchmark tests, the DMIPS numbers are inversely proportional to the simulation run time. Further, in the Redback case DMIPS/MHz is calculated using the instruction fetch count and so has higher DMIPS/MHz numbers than the synchronous designs.

5.6 Summary

This chapter has described the design and performance of the Redback RISC core– the NCL based RISC-V CPU. Initially, the RISC-V ISA used for this core was presented, and then all of the sub-modules and their interfaces discussed and analysed. These sub-modules plus the top core were verified as correct using the RISC-V verification programs from UC-Berkeley. The core was also tested on an Altera/Intel FPGA board after implementation on the FPGA synthesis tools using the FPGA library previously discussed in chapter 3.4. The CPU activity in response to a number of benchmark tests was monitored using the built-in FPGA signal monitoring program. The remainder of the chapter shows the CPU optimization methodology and the design flow for the comparison purposes. A statistical analysis of the dynamic instruction count was performed for each benchmark as a first step towards further performance optimization. Unlike synchronous CPU design, this profiling approach will be important for the optimization of asynchronous CPU system.

			PicoRV32	PicoRV32 UNCLE	Rocket-Tiny	Rocket-Tiny UNCLE	Redback
Clock Frequency			1.5GHz		1GHz		371MIPS
Simulation Run Time	(nS)		472,393		281,994		546,264
Area	Cell Count	Total Cells	7,954	50,094	7,712	69,494	22,892
(μm^2)		Combinational Cells	6,051		6,116		
		Sequential Cells	1,855		1,550		
	Cell Area		10,204	78,695	12,768	109,896	32,536
Average Power	Total Power		8.793		2.929		5.038
(mW)	Dynamic		8.589		2.664		5.006
	Static		0.205		0.264		0.033
DMIPS			602		1,009		521
DMIPS/MHz			0.402		1.009		1.404

Table 30: RISC-V vs. RV32IM Core Benchmark Test Comparison

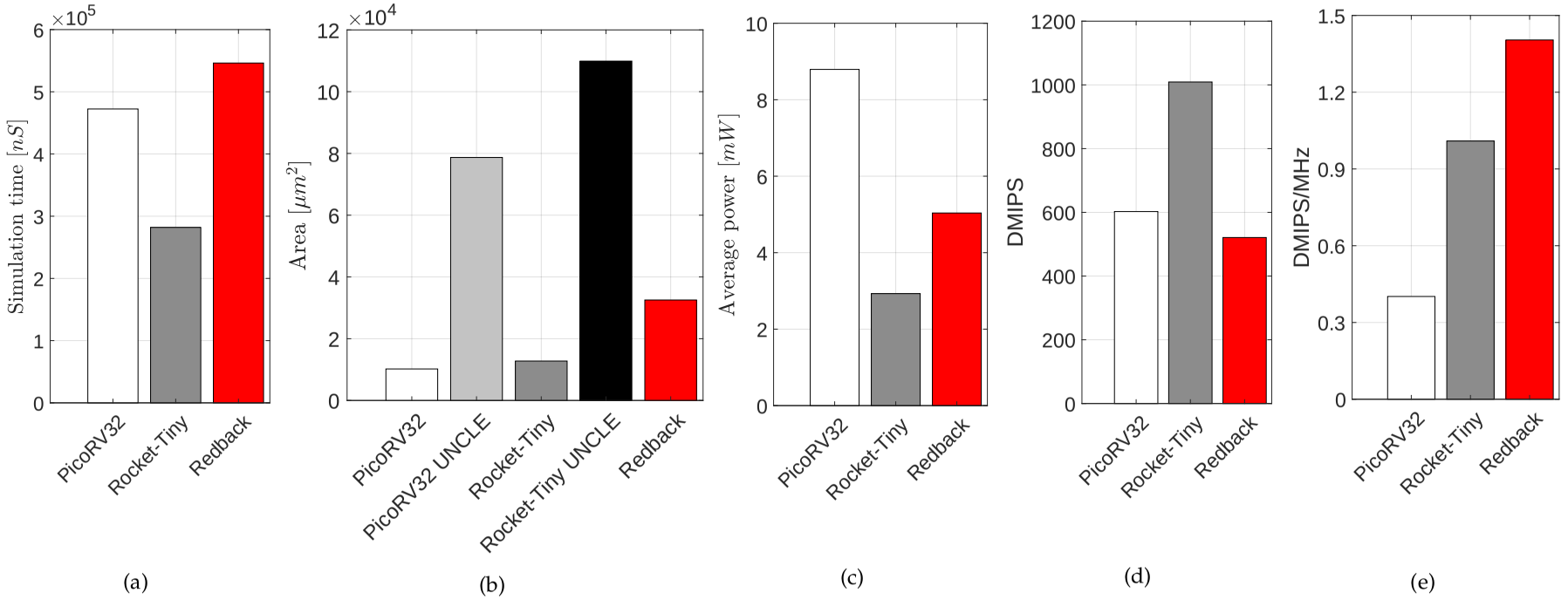

Figure 143: Comparison between participated methods in terms of:
(a) Simulated Run Time, (b) Area, (c) Average Power, (d) DMIPS, and (e) DMIPS/MHz

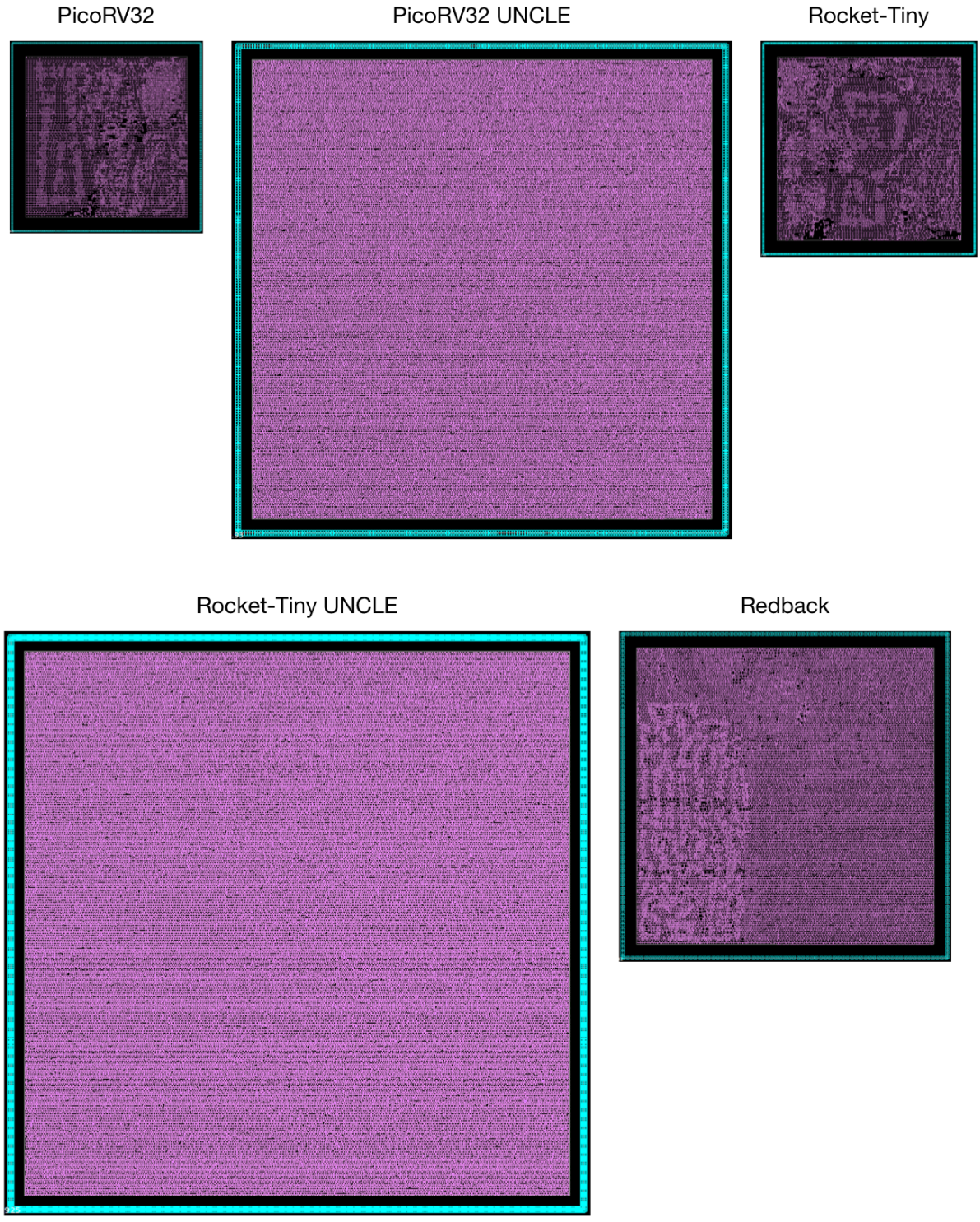

Figure 144: RISC-V RV32IM Core Area Comparison (Post-P&R)

Just as for the simple examples presented in Chapter 3, three alternative approaches and their flows have been explored for the design of the CPU core - synchronous, UNCLE and NELL and data for the benchmark tests, comparing their performance, power and area, have been presented and analysed.

The comparison results show the advantage of the NELL based design compared to the UNCLE automatic conversion tools, reinforcing the decision made as a result of the preliminary experiments. However, it is clear that manual decomposition, such as can be performed using System-Verilog, produces the most efficient result. However, much more work is required to produce circuits that compete with manual techniques.

As identified in [7], it is possible to reuse synchronous synthesis tools to translate a synchronous gate-level net-list into an equivalent asynchronous net-list. This is more-or-less what the UNCLE tools are doing and this flow has the benefits of reducing the barriers to the adoption of asynchronous design.

The tool flows developed for this work are mostly based on current synchronous ASIC implementation tools such as simulation, place and route and timing/power analysis tools. As these tools are designed for synchronous digital implementation they have limited support for asynchronous design.

The case study in this chapter using the NELL environment clearly shows that a tool optimised for NCL design can achieve a much better result than the simplistic translation method employed by UNCLE. Once dedicated and efficient asynchronous design tools are developed, it should be possible to compete more successfully with synchronous designs, especially for complex designs such as this 32-bit embedded CPU.

The comparison results illustrate the viability of this 32-bit asynchronous CPU implementation. Additionally, asynchronous design styles have useful dynamic supply voltage scaling capabilities, which can offer additional power/energy advantages especially for sensitive applications such as battery operated mobile devices and IoT applications [150]. Achieving lower supply voltages such as near-threshold or sub-threshold voltages would require a well designed NCL cell library targeting CPU design in particular. Because most NCL cells exhibit hysteresis, extensive static noise margin simulations at lower supply voltages will be essential

for those implementations. This is outside the scope of this thesis but represents important future work to support asynchronous CPU research.

From the results presented here, we can see that area is the biggest drawback for NCL based asynchronous designs, particularly in the case of the register file and multiplier blocks. These have extremely large gate counts and thus their cell area is very large compared to the corresponding synchronous circuit. It may be possible to optimize the cell count and area by adding dedicated custom cells aimed specifically at the CPU components. Examples of such optimised cells would be 4:2 Compressors and Booth Encoder cells for the Multiplier plus dedicated NCL Register File cells for the NCL Register File block.

Overall, this implementation of the *Redback RISC* demonstrates that complex CPU design is possible using NCL, and serves to motivate future research into asynchronous CPU techniques and tools.

Chapter 6

Summary, Conclusions and Future Work

In the previous chapters, the background research and literature review and design methodology of Null Convention Logic were presented. The detailed circuit designs for the NCL CPU core components were introduced and the Redback RISC CPU core implemented and compared with corresponding synchronous designs. In this Chapter, we summarize this research, highlight its contributions and conclude the work. Finally, some suggestions for additional work arising from the research are presented.

6.1 Summary

A Null Convention Logic based asynchronous 32-bit CPU core (called the *Redback RISC*) has been designed, analysed and demonstrated in this thesis. This research commenced with a study of the background to asynchronous technology and their design methodologies. There are three major asynchronous design techniques, Bundled-Data, QDI and NCL, and we selected NCL for our design. The detailed design methodologies were studied, particularly the techniques for implementing NCL designs on ASIC and FPGA. Many tools and synthesis techniques have been used to explore the NCL cells and modules, including transistor-level schematic entry, transistor layout using standard foundry templates and DRC/LVS, parasitic extraction, Spice simulation and verification. Synthesis, Auto Place and Route tools were used for CPU-level design and comparison, as well as sign-off timing and power analysis tools for implementation and result analysis. Standard digital verification simulation tools have been used to simulate all of the arithmetic modules and CPU core blocks and the entire CPU core has been simulated with System Verilog test-benches and compiled program binary images. Using

the 28nm FDSOI deep sub-micron technology, we could test and analyze the asynchronous circuit performance with the latest technology. Xilinx, Altera/Intel and Actel (Microsemi) FPGA devices/boards and their implementation tools were used for the FPGA prototyping. Performing these simulations allowed validation of the asynchronous circuits but also revealed other aspects of the NCL performance such as the occurrence of dead-locks, orphans and the like.

The Redback RISC CPU core was tested on the FPGA board with benchmark test programs that were compiled using the RISC-V compiler. The NELL programming language was mainly used for design in this work. NELL is a dedicated programming language and compiler environment for NCL. We also used UNCLE to convert synchronous circuits to NCL, mainly for comparison and testing purposes. Because UNCLE is not a commercial tool, its correct operation requires additional automation scripts that were built as part of this work. While NELL proved to result in better implementations, to program using NELL requires a deeper understanding of NCL technology and its design skills compared to UNCLE, which needs more understanding of schematic level NCL circuit designs.

This work implemented the CPU core using the latest RISC-V open instruction set architecture[1]. The entire 32-Bit RISC-V core was built using pure NCL technology without any clock or clocked devices. All the detailed arithmetic devices were generated and compared especially many adder and multiplier types. The arithmetic blocks have been logically verified against their synchronous counter parts. Of all the CPU blocks, the Program Counter and Register File were the most complex and challenging. The Program Counter ring gives *liveness* to the whole CPU while the main challenge in the Register File is its separated write and read ports. Because the NCL handshaking is initiated from the register file read ports and terminated at the register file write ports, appropriate handshaking controls were required. Especially in the results writing case, the registers must be initialized to NULL before any DATA values could be written to that Register.

The Redback RISC core was compared with two standard synchronous CPU core designs, the PicoRV32 and the Rocket with Tiny configuration. These synchronous cores are the most popular 32-Bit cores in the RISC-V community and are used for chip fabrication in industry. Using the Dhrystone benchmark tests, their performance, area and power were compared.

[1]The RISC-V community is very active and were very supportive of this research and provided a great compile/test environment.

6.2 Conclusions

This research encompasses a study of the characteristics and performance of Null Convention Logic and an analysis of the RISC-V ISA and CPU architecture including its component blocks, in particular those applicable to a 32-bit CPU core. NCL technology is rarely applied to high performance CPU cores, especially for 32-bit Embedded Processors. Previous NCL based CPU designs were mostly based on 8-bit ISAs such as 8051 but the demand for 32-bit architectures is increasing substantially because the Internet of Things(IoT) is becoming widespread, in turn requiring CPU cores with larger memory and higher performance, while at the same time operating with low energy.

This research has resulted in the development of a 32-Bit asynchronous RISC-V CPU core called **Redback RISC** that has been compared to two approximately equivalent industry standard 32-bit synchronous cores. As far as we are aware, this is the first time NCL technology has been applied to the design of a 32-Bit CPU core.

Further, the implementation results were also compared with results using an existing NCL design tool (UNCLE), which showed how much the results of these implementation strategies differ. The Redback RISC has achieved similar levels of throughput and 43% better power and 34% better energy compared to one of the synchronous cores with the same benchmark tests and test conditions such as input supply voltage. However, it was shown that area is the biggest drawback for NCL CPU design. The core is roughly 2.5× larger than the synchronous designs. On the other hand the description using NELL has resulted an area that is 2.9× smaller than an implementation using the UNCLE tools.

The Redback RISC core and many of the component arithmetic blocks were successfully validated on a small number of commercial FPGA devices and a design and test methodology has been developed for this process. FPGA prototyping is the one of the biggest barrier for asynchronous designs and this methodology has made that easier and clearly showed how commercial FPGA can be used to support NCL circuits. A number of benchmark tests were demonstrated on FPGA devices using the Redback RISC.

The high performance multi-dimensional design approach, especially for the high speed 32 x 32 multiplier is a major challenge in the NCL design domain. High speed multipliers are the basic building block for many application designs such as Finite Impulse Response (FIR) Filters or Fast Fourier Transform (FFT) circuits and also Flow Graph designs like

streaming media processing (Video and Audio), Ethernet Packet processing and Cryptography circuits and so on. This research has clearly shown the advantages and trade-offs when using multi-dimensional (e.g., 2D) NCL designs. Future designers will be able to reuse these results when they consider high performance NCL circuit especially for the Data Flow designs.

A Register File Write-Back Queue design has been proposed in this work and the trade-off between NCL based FIFO and Data-Queue structures shown in terms of performance, power and area. Using these comparison results, designers can select the appropriate buffer types for other processor blocks in the NCL based asynchronous microprocessor and system on chip design.

Even though NCL has many technical advantages compared to conventional clocked designs and other asynchronous approaches, much effort is still required to optimize these NCL circuits. The approach described in this work, encompassing both structural level and circuit level optimizations of the NCL designs, illustrates a concrete methodology for NCL circuit design with detailed applications such as high speed 32x32 multiplier and 32-Bit high speed microprocessor. This represents a useful design guide for future students and engineers who may wish to apply NCL technology.

6.3 Future Work

The process of developing this CPU core has thrown up many areas of important future work. A few of the more important issues are discussed in the following sections.

6.3.1 Voltage scaling test and cell library design

As we discussed at the conclusion of Chapter 5, compared to conventional synchronous design, NCL shows its biggest benefit with low supply voltages, especially sub-threshold/near-threshold because the circuit design style is more able to cope with Process, Voltage and Temperature variations. To achieve that, we need well designed NCL cell library suited to low supply voltage operation. The cells have to have high noise margin across the range of variation, because most NCL cells have internal hysteresis that can misbehave in the presence of strong noise. With this voltage scaling ability, we can get optimum benefit from the asynchronous processor design methodology.

6.3.2 Automated NCL timing constraints and timing driven place & route

Even though we could reuse most of the synchronous physical implementation tools for NCL design, the tools still have a lot of limitations which prevent the efficient implementation of NCL designs. The timing constraints of NCL are totally different from the worst case behavior of synchronous timing therefore in NCL we have to use statistical approaches base on its average case performance. To do that we need to investigate the activity statistics of each different execution module and the instruction decoder. The activity statistics are very dependant on the real application software and application requirements. We need to develop automation software that would support this approach and allow the generation of complete timing constraints and their implementation priorities for use by the Back-end Place and Route tools.

6.3.3 High speed NCL arithmetic modules

In Asynchronous digital design, especially CPU design, we have to have different approach from the conventional synchronous design. We can design Adders and Multipliers etc. with high speed independent rings. Of course, synchronous design still can be implemented with different clock speeds for each arithmetic sub-modules or with separate clock domains but it will be much easier using asynchronous techniques which can be implemented without glue logic such as FIFOs, or without needing Clock Boundary Crossing circuits. In synchronous design, to implement high speed arithmetic modules requires special clock tree structures, and thus will have some clock tree PVT variation issues. It can require a lot of resources to create high speed clock trees. In contrast this is unlikely to be a problem for asynchronous systems. The availability of high speed NCL arithmetic modules with the independent high speed rings would greatly benefits the NCL design process.

6.3.4 ISA binary translation for the Asynchronous RISC-V CPU

Many of the previous asynchronous CPU design have used their own instruction sets, because the conventional ISA for the synchronous design is not a good fit to the average case, delay insensitive CPU core design. We used the RISC-V ISA, which is simple and sophisticated but it still not fully suited to asynchronous designs. For example, the pseudo-op strategy of RISC-V is not ideal for NCL in that many of the ops use adders as data busses (J instruction) or exercise the ALU and the register file to just do nothing (NOP). All this switching activity can be avoided in an NCL design. We suggest ISA binary translation for the future work and

convert this RISC-V ISA to the dedicated asynchronous CPU design. Therefore we still can use the same compile environment while further optimising the performance of the RISC-V machine.

References

[1] C. Weltin-Wu and Y. Tsividis, "An event-driven clockless level-crossing ADC with signal-dependent adaptive resolution", *IEEE Journal of Solid-State Circuits*, vol. 48, no. 9, pp. 2180–2190, 2013.

[2] A. J. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings, and T. K. Lee, "The design of an asynchronous MIPS R3000 microprocessor", in *Proceedings Seventeenth Conference on Advanced Research in VLSI*, 1997, pp. 164–181.

[3] R. B. Reese, S. C. Smith, and M. A. Thornton, "Uncle - an RTL approach to asynchronous design", in *18th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2012, pp. 65–72, ISBN: 1522-8681.

[4] Wave Computing. (2019). Wave Computing, [Online]. Available: `https://wavecomp.ai`.

[5] R.-V. Foundation. (). 2nd risc-v workshop, [Online]. Available: `https://riscv.org/2015/07/2nd-risc-v-workshop/`.

[6] J. Sparso and S. Furber, *Principles Asynchronous Circuit Design*. Springer, 2002, ISBN: 0792376137.

[7] P. A. Beerel, R. O. Ozdag, and M. Ferretti, *A Designer's Guide to Asynchronous VLSI*. Cambridge ; New York: Cambridge UP, 2010. Print, 2010, ISBN: 9780521872447.

[8] Wiki. (2019). synchronous Circuit, [Online]. Available: `https://en.wikipedia.org/wiki/Asynchronous_circuit`.

[9] Achronix. (2019), [Online]. Available: `http://www.achronix.com/`.

[10] A. Lines, P. Joshi, R. Liu, S. McCoy, J. Tse, Y. Weng, and M. Davies, "Loihi asynchronous neuromorphic research chip", in *2018 24th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2018, pp. 32–33.

[11] Tiempo, *http://www.tiempo-secure.com/*,

[12] E. Compute. (). Etacore, [Online]. Available: `https://etacompute.com/`.

[13] J. Sparsø, "Current trends in high-level synthesis of asynchronous circuits", in *2009 16th IEEE International Conference on Electronics, Circuits and Systems - (ICECS 2009)*, 2009, pp. 347–350.

[14] A. Bardsley and D. A. Edwards, "Balsa: An asynchronous circuit synthesis system", The University of Manchester, Tech. Rep., 1999.

[15] R. B. Reese, "Uncle (unified NCL environment)", Technical Report MSU-ECE-10-001 November 2010, available online: http://www.ece.msstate.edu/ reese/uncle/UNCLE.pdf, Report, 2011.

[16] P. A. Beerel, G. D. Dimou, and A. M. Lines, "Proteus: An ASIC flow for ghz asynchronous designs", *IEEE Design Test of Computers*, vol. 28, no. 5, pp. 36–51, 2011.

[17] I. E. Sutherland, "Micropipelines", *Communications of the ACM*, vol. 32, no. 6, pp. 720–738, 1989.

[18] K. M. Fant, *Logically determined design: clockless system design with NULL convention logic*. John Wiley & Sons, 2005, ISBN: 0471702870.

[19] S. C. Smith and J. Di, "Designing asynchronous circuits using NULL convention logic (NCL)", *Synthesis Lectures on Digital Circuits and Systems*, vol. 4, no. 1, pp. 1–96, 2009.

[20] K. M. Fant and S. A. Brandt, *Null convention logic system*, US Patent 5305463, 1994.

[21] ——, "Null convention logic: A complete and consistent logic for asynchronous digital circuit synthesis", in *Proceedings of the International Conference on Application Specific Systems, Architectures and Processors, ASAP'96*, 1996, pp. 261–273, ISBN: 2160-0511.

[22] K. Fant. (2019). Theseus research, [Online]. Available: http://www.theseusresearch.com/.

[23] S. C. Smith, "Design of an FPGA logic element for implementing asynchronous null convention logic circuits", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 6, pp. 672–683, 2007.

[24] F. A. Parsan and S. C. Smith, "CMOS implementation of static threshold gates with hysteresis: A new approach", in *20th International Conference on VLSI and System-on-Chip (VLSI-SoC)*, IEEE/IFIP, 2012, pp. 41–45, ISBN: 1467326577.

[25] G. E. Sobelman and K. Fant, "CMOS circuit design of threshold gates with hysteresis", in *IEEE International Symposium on Circuits and Systems*, IEEE, 1998, pp. 61–64, ISBN: 0271-4310.

[26] P.-Q. Cuong and D.-D. Anh-Vu, "Hazard-free muller gates for implementing asynchronous circuits on xilinx fpga", in *Fifth IEEE International Symposium on Electronic Design, Test and Application, DELTA'10*, 2010, pp. 289–292.

[27] K. Maheswaran and V. Akella, "Hazard-free implementation of the self-timed cell set in a Xilinx FPGA", *Tech. Report: University of California Davis, CA*, 1994.

[28] L. Yijun, X. Guobo, C. Pinghua, C. Jingyu, and L. Zhenkun, "Designing an asynchronous FPGA processor for low-power sensor networks", in *International Symposium on Signals, Circuits and Systems, ISSCS*, 2009, pp. 1–6.

[29] D. L. Oliveira, S. S. Sato, O. Saotome, and R. T. de Carvalho, "Hazard-free implementation of the extended burst-mode asynchronous controllers in look-up table based fpga", in *4th Southern Conference on Programmable Logic*, 2008, pp. 143–148.

[30] Q. T. Ho, J.-B. Rigaud, L. Fesquet, M. Renaudin, and R. Rolland, "Implementing asynchronous circuits on LUT based FPGAs", in *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*. Springer, 2002, pp. 36–46, ISBN: 3540441085.

[31] P. D. Ferguson, A. Efthymiou, T. Arslan, and D. Hume, "Optimising self-timed fpga circuits", in *13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD)*, 2010, pp. 563–570.

[32] J. Wu, Y. Shi, and M. Choi, "FPGA-based measurement and evaluation of power analysis attack resistant asynchronous s-box", in *IEEE Instrumentation and Measurement Technology Conference (I2MTC)*, IEEE, 2011, pp. 1–6, ISBN: 1424479339.

[33] E. Brunvand, "Using fpgas to implement self-timed systems", *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 6, no. 2, pp. 173–190, 1993.

[34] K. Meekins, D. Ferguson, and M. Basta, "Delay insensitive NCL reconfigurable logic", in *IEEE Aerospace Conference Proceedings*, vol. 4, 2002, 4–1961–4–1966 vol.4.

[35] S. C. Smith, "Design of a logic element for implementing an asynchronous FPGA", in *Proceedings of the 15th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '07, Monterey, California, USA: ACM, 2007, pp. 13–22, ISBN: 978-1-59593-600-4.

[36] J. Yu and P. Beckett, "A dual-rail LUT for reconfigurable logic using null convention logic", in *Proceedings of the 24th Edition of the Great Lakes Symposium on VLSI*, ser. GLSVLSI '14, Houston, Texas, USA: ACM, 2014, pp. 261–266, ISBN: 978-1-4503-2816-6.

[37] S. Ramaswamy, L. Rockett, D. Patel, S. Danziger, R. Manohar, C. W. Kelly, J. L. Holt, V. Ekanayake, and D. Elftmann, "A radiation hardened reconfigurable FPGA", in *2009 IEEE Aerospace conference*, pp. 1–10.

[38] R. E Meagher and I. P Nash, "The ordvac [includes discussion]", eng, in *1951 International Workshop on Managing Requirements Knowledge*, IEEE, 1951, pp. 37–37.

[39] ETHW. (). WEIZAC computer, [Online]. Available: https://ethw.org/Milestones:WEIZAC_Computer,_1955.

[40] H. C Brearley, "ILLIAC II-A short description and annotated bibliography", eng, *IEEE Transactions on Electronic Computers*, vol. EC-14, no. 3, pp. 399–403, 1965.

[41] Atlas Computer Laboratory. (2019). Atlas computer, [Online]. Available: `ttp://www.chilton-computing.org.uk/acl/technology/atlas/p019.htm`.

[42] Wiki. (2019). Atlas computer, [Online]. Available: `https://en.wikipedia.org/wiki/Atlas_(computer)`.

[43] H. W. Lawson Jr and B. Malm, "A flexible asynchronous microprocessor", *BIT Numerical Mathematics*, vol. 13, no. 2, pp. 165–176, 1973.

[44] R. A. Lorie and H. R. Strong. (Apr. 1984). Weak synchronization and scheduling among concurrent asynchronous processors. US Patent 4445197A, [Online]. Available: `https://patents.google.com/patent/US4445197A/fr`.

[45] M. J. Cochran, *Asynchronous high speed processor having high speed memories with domino circuits contained therein*, US Patent 4680701A, Apr. 1984.

[46] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus, "The design of an asynchronous microprocessor", *SIGARCH Computer Architecture News*, vol. 17, no. 4, pp. 99–110, Jun. 1989.

[47] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus, "The first asynchronous microprocessor: The test results", *SIGARCH Computer Architecture News*, vol. 17, no. 4, pp. 95–98, Jun. 1989.

[48] K. Shinji, "The data-driven microprocessor", *IEEE Micro*, vol. 9, no. 3, pp. 45–59, 1989.

[49] G. M. Jacobs and R. W. Brodersen, "A fully asynchronous digital signal processor using self-timed circuits", *IEEE Journal of Solid-State Circuits*, vol. 25, no. 6, pp. 1526–1537, 1990.

[50] R. Ginosar and N. Michell, "On the potential of asynchronous pipelined processors", *ACM SIGARCH Computer Architecture News*, vol. 18, no. 4, pp. 27–34, 1990.

[51] I. David, R. Ginosar, and M. Yoeli, "Self-timed architecture of a reduced instruction set computer", *Asynchronous Design Methodologies*, vol. 28, pp. 29–43, 1993.

[52] K.-R. Cho, K. Okura, and K. Asada, "Design of a 32-bit fully asynchronous microprocessor (fam)", in *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, IEEE, 1992, pp. 1500–1503, ISBN: 0780305108.

[53] N. Paver, P. Day, S. B. Furber, J. D. Garside, and J. V. Woods, "Register locking in an asynchronous microprocessor", in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD'92*, IEEE, 1992, pp. 351–355, ISBN: 0818631104.

[54] J. D. Garside, "A CMOS VLSI implementation of an asynchronous ALU", in *Proceedings of the IFIP WG10.5 Working Conference on Asynchronous Design Methodologies*, Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 1993, pp. 181–192, ISBN: 0-444-81599-6.

[55] S. B. Furber, P. Day, J. D. Garside, N. Paver, and J. V. Woods, "A micropipelined ARM", 1993.

[56] M. E. Dean, "STRiP: a self-timed RISC processor", PhD Thesis, 1992.

[57] A. Costa, A. De Gloria, P. Faraboschi, G. Nateri, and M. Olivieri, "An asynchronous approach to the RISC design of a micro-controller", *Microprocessing and Microprogramming*, vol. 38, no. 1, pp. 447–454, 1993.

[58] A. D. Gloria, P. Faraboschi, and M. Olivieri, "A self timed interrupt controller: A case study in asynchronous micro-architecture design", in *Proceedings of the Seventh Annual IEEE International ASIC Conference and Exhibit*, IEEE, 1994, pp. 296–299, ISBN: 0780320204.

[59] T. Nanya, "Challenges to dependable asynchronous processor design", in *Logic Synthesis and Optimization*. Springer, 1993, pp. 191–213, ISBN: 1461363810.

[60] E. Brunvand, "The nsr processor", in *Proceedings of the Twenty-Sixth Hawaii International Conference on System Sciences*, vol. 1, IEEE, 1993, pp. 428–435, ISBN: 0818632305.

[61] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, S. Temple, and J. V. Woods, "The design and evaluation of an asynchronous microprocessor", in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD'94*, IEEE, 1994, pp. 217–220, ISBN: 0818665653.

[62] J. A. Tierno, A. J. Martin, D. Borkovic, and T. K. Lee, "A 100-MIPS GaAs asynchronous microprocessor", *IEEE Design and Test of Computers*, vol. 11, no. 2, pp. 43–49, 1994.

[63] S. V. Morton, S. S. Appleton, and M. J. Liebelt, "ECSTAC: a fast asynchronous microprocessor", in *Proceedings of the Second Working Conference on Asynchronous Design Methodologies*, Jun. 1995, pp. 180–189.

[64] W. F. Richardson and E. Brunvand, "Fred: An architecture for a self-timed decoupled computer", in *Proceedings Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1996, pp. 60–68.

[65] P. B. Endecott, "SCALP: a superscalar asynchronous low-power processor", Thesis, 1996.

[66] J. Tse and D. P. Lun, "ASYNMPU: a fully asynchronous CISC microprocessor", in *Proceedings of 1997 IEEE International Symposium on Circuits and Systems, ISCAS'97*, vol. 3, IEEE, 1997, pp. 1816–1819, ISBN: 078033583X.

[67] A. Semenov, A. M. Koelmans, L. Lloyd, and A. Yakovlev, "Designing an asynchronous processor using petri nets", *IEEE Micro*, no. 2, pp. 54–64, 1997.

[68] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, and N. C. Paver, "Amulet2e: An asynchronous embedded controller", in *Proceedings of the Third International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1997, pp. 290–299.

[69] J. D. Garside, S. Temple, and R. Mehra, "The amulet2e cache system", in *Proceedings of the Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1996, pp. 208–217.

[70] S. B. Furber, J. D. Garside, and D. A. Gilbert, "Amulet3: A high-performance self-timed arm microprocessor", in *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors, ICCD'98*, IEEE, 1998, pp. 247–252, ISBN: 0818690992.

[71] A. Takamura, M. Kuwako, M. Imai, T. Fujii, M. Ozawa, I. Fukasaku, Y. Ueno, and T. Nanya, "TITAC-2: An asynchronous 32-bit microprocessor based on scalable-delay-insensitive model", in *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors, ICCD'97*, IEEE, 1997, pp. 288–294, ISBN: 081868206X.

[72] A. Takamura, M. Imai, M. Ozawa, I. Fukasaku, T. Fujii, M. Kuwako, Y. Ueno, and T. Nanya, "Titac-2: An asynchronous 32-bit microprocessor", in *Design Automation Conference 1998. Proceedings of the ASP-DAC'98. Asia and South Pacific*, IEEE, pp. 319–320, ISBN: 0780344251.

[73] R. Kol and R. Ginosar, "Kin: A high performance asynchronous processor architecture", in *Proceedings of the 12th international conference on Supercomputing*, ACM, pp. 433–440, ISBN: 089791998X.

[74] H. Van Gageldonk, K. Van Berkel, A. Peeters, D. Baumann, D. Gloor, and G. Stegmann, "An asynchronous low-power 80c51 microcontroller", in *Proceedings of the Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, IEEE, 1998, pp. 96–107, ISBN: 0818683929.

[75] M. Renaudin, P. Vivet, and F. Robin, "ASPRO-216: A standard-cell QDI 16-bit RISC asynchronous microprocessor", in *Proceedings of the Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, IEEE, 1998, pp. 22–31, ISBN: 0818683929.

[76] K. T. Christensen, P. Jensen, P. Korger, and J Sparso, "The design of an asynchronous TinyRISC TR4101 microprocessor core", in *Proceedings of the Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, IEEE, 1998, pp. 108–119, ISBN: 0818683929.

[77] S. S. Appleton, "Performance-directed design of asynchronous vlsi systems", Thesis, 1997.

[78] J. D. Garside, W. J. Bainbridge, A. Bardsley, D. M. Clark, D. A. Edwards, S. B. Furber, J. Liu, D. W. Lloyd, S. Mohammadi, J. S. Pepper, O. Petlin, S. Temple, and J. V. Woods, "Amulet3i-an asynchronous system-on-chip", in *Proceedings Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 2000) (Cat. No. PR00586)*, Apr. 2000, pp. 162–175.

[79] Je-Hoon Lee, Won-Chul Lee, and Kyoung-Rok Cho, "A novel asynchronous pipeline architecture for CISC type embedded controller, A8051", in *The 2002 45th Midwest Symposium on Circuits and Systems, 2002. MWSCAS-2002*, vol. 2, Aug. 2002, pp. II–II.

[80] J. Lee, Y. H. Kim, and K. Cho, "A low-power implementation of asynchronous 8051 employing adaptive pipeline structure", *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 55, no. 7, pp. 673–677, Jul. 2008.

[81] L. A. Plana, P. A. Riocreux, W. J. Bainbridge, A. Bardsley, J. D. Garside, and S. Temple, "SPA - a synthesisable amulet core for smartcard applications", in *Proceedings Eighth International Symposium on Asynchronous Circuits and Systems*, Apr. 2002, pp. 201–210.

[82] M. Oh, Y. W. Kim, S. Kwak, C. Shin, and S. Kim, "Architectural design issues in a clockless 32-bit processor using an asynchronous hdl", *ETRI Journal*, vol. 35, no. 3, pp. 480–490, Jun. 2013.

[83] A. J. Martin, M. Nystrom, K. Papadantonakis, P. I. Penzes, P. Prakash, C. G. Wong, J. Chang, K. S. Ko, B. Lee, E. Ou, J. Pugh, E. Talvala, J. T. Tong, and A. Tura, "The Lutonium: A sub-nanojoule asynchronous 8051 microcontroller", in *Ninth International Symposium on Asynchronous Circuits and Systems, 2003. Proceedings.*, May 2003, pp. 14–23.

[84] C. Kelly, V. Ekanayake, and R. Manohar, "SNAP: A sensor-network asynchronous processor", in *Ninth International Symposium on Asynchronous Circuits and Systems, 2003. Proceedings.*, May 2003, pp. 24–33.

[85] H. Solutions. (). Ht80c51, [Online]. Available: `http://www.keil.com/dd/chip/3931.htm`.

[86] A. Bink and R. York, "ARM996HS™ the first licensable, clockless 32-bit processor core", in *2006 IEEE Hot Chips 18 Symposium (HCS)*, Jul. 2006, pp. 1–28.

[87] A. Lines, "The Vortex: a superscalar asynchronous processor", in *13th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'07)*, Mar. 2007, pp. 39–48.

[88] J.-H. Lee and K.-R. Cho, "Design of a high performance self-timed arm9 processor", *IEICE Electronics Express*, vol. 5, no. 3, pp. 87–93, 2008.

[89] B. Hollosi, M. Barlow, Guoyuan Fu, C. Lee, Jia Di, S. C. Smith, H. A. Mantooth, and M. Schupbach, "Delay-insensitive asynchronous ALU for cryogenic temperature environments", in *2008 51st Midwest Symposium on Circuits and Systems*, Jul. 2008, pp. 322–325.

[90] B. Hollosi. (2008). 8051-compliant asynchronous microcontroller core design, fabrication, and testing for extreme environment, PhD thesis, ProQuest Dissertations and Theses. English, [Online]. Available: `https://search.proquest.com/docview/304687776/fulltextPDF/54B19C36C15C459DPQ/`.

[91] Tiempo. (). Tam16, [Online]. Available: `http://www.tiempo-ic.com/products/TAM16.html`.

[92] S. Kwak, H. Lee, Y. Zafar, M. Oh, and D. Har, "Design of asynchronous MSP430 microprocessor using balsa back-end retargeting", in *2009 5th Southern Conference on Programmable Logic (SPL)*, Apr. 2009, pp. 223–228.

[93] S. Keller, A. J. Martin, and C. Moore, "DD1: A QDI, Radiation-Hard-by-Design, Near-Threshold 18uW/MIPS Microcontroller in 40nm Bulk CMOS", in *2015 21st IEEE International Symposium on Asynchronous Circuits and Systems*, May 2015, pp. 37–44.

[94] D. Hand, M. T. Moreira, H. Huang, D. Chen, F. Butzke, Z. Li, M. Gibiluka, M. Breuer, N. L. V. Calazans, and P. A. Beerel, "Blade – a timing violation resilient asynchronous template", in *2015 21st IEEE International Symposium on Asynchronous Circuits and Systems*, 2015, pp. 21–28.

[95] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha, "Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1537–1557, 2015.

[96] M. Davies, N. Srinivasa, T. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y. Weng, A. Wild, Y. Yang, and H. Wang, "Loihi: A neuromorphic manycore processor with on-chip learning", *IEEE Micro*, vol. 38, no. 1, pp. 82–99, Jan. 2018.

[97] A. D. C. Inc., *Instruction set reference manual for AE32000: a 32-bit EISC microprocessor*, Nov. 2008.

[98] Texas Instruments. (2019). MSP430 family instruction set summary, [Online]. Available: `https://www.ti.com/sc/docs/products/micro/msp430/userguid/as_5.pdf`.

[99] RISC-V Foundation. (). RISC-V ISA, [Online]. Available: `https://riscv.org/risc-v-isa/`.

[100] A. Waterman, "Improving energy efficiency and reducing code size with risc-v compressed", Master's thesis, EECS Department, University of California, Berkeley, May 2011.

[101] C. Jeong, *Optimization techniques for robust asynchronous threshold networks*, eng, 2008.

[102] CMP. (2019). 28nm FDSOI, [Online]. Available: `https://mycmp.fr/datasheet/ic-28nm-cmos28fdsoi`.

[103] S. Yancey and S. C. Smith, "A differential design for c-elements and ncl gates", in *2010 53rd IEEE International Midwest Symposium on Circuits and Systems*, 2010, pp. 632–635.

[104] H. J. Lee and Y. Kim, "Low power null convention logic circuit design based on dcvsl", in *2013 IEEE 56th International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2013, pp. 29–32.

[105] J. Kim, M. M. Kim, and P. Beckett, "Static leakage control in null convention logic standard cells in 28 nm utbb-fdsoi cmos", in *2015 International SoC Design Conference (ISOCC)*, 2015, pp. 99–100.

[106] M. Kim. (). Ncl gate functional simulation, [Online]. Available: `https://github.com/MatthewMyunghaKim/NCL_Gates_Functional_Simulation`.

[107] M. M. Kim and P. Beckett, "Design techniques for ncl-based asynchronous circuits on commercial fpga", in *2014 17th Euromicro Conference on Digital System Design*, Jul. 2014, pp. 451–458.

[108] S. C. Smith, W. K. Al-Assadi, and J. Di, "Integrating asynchronous digital design into the computer engineering curriculum", *IEEE Transactions on Education*, vol. 53, no. 3, pp. 349–357, 2010.

[109] D. A. Duncan, G. E. Sobelman, and K. M. Fant, *Null convention adder*, U.S. Patent 5793662A, June 1993.

[110] K. Makoto, *High speed adder circuit*, US Patent 3566098A, 1969.

[111] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations", eng, *IEEE Transactions on Computers*, vol. C-22, no. 8, pp. 786–793, 1973.

[112] Brent and Kung, "A regular layout for parallel adders", eng, *IEEE Transactions on Computers*, vol. C-31, no. 3, pp. 260–264, 1982.
[113] T. Han and D. A. Carlson, "Fast area-efficient vlsi adders", eng, in *1987 IEEE 8th Symposium on Computer Arithmetic (ARITH)*, IEEE, 1987, pp. 49–56, ISBN: 0818607742.
[114] R. E. Ladner and M. J. Fischer, "Parallel prefix computation", *J. ACM*, vol. 27, no. 4, pp. 831–838, Oct. 1980.
[115] O. J Bedrij, "Carry-select adder", eng, *IRE Transactions on Electronic Computers*, vol. EC-11, no. 3, pp. 340–346, 1962.
[116] J. Sklansky, "Conditional-sum addition logic", *IRE Transactions on Electronic Computers*, vol. EC-9, no. 2, pp. 226–231, 1960.
[117] Y. Kobayashi, A. Satoh, and S. Munetoh, *Carry skip adder*, U.S. Patent 6735612, May 2004.
[118] J. G. Earle, *Latched carry save adder circuit for multipliers*, U.S. Patent 3340388A, JUL. 1965.
[119] P. Blankenship, "Comments on "a two's complement parallel array multiplication algorithm"", eng, *IEEE Transactions on Computers*, vol. C-23, no. 12, pp. 1327–1327, 1974.
[120] D. R. Kondu and P. V. Mahesh, "Design and analysis of 32x32 bit alu using high-speed vedic-wallace multiplier based on vedic mathematics", English, *i-Manager's Journal on Electronics Engineering*, vol. 6, no. 3, pp. 7–14, Mar. 2016, Copyright - 2016 i-manager publications. All rights reserved; Document feature - ; Last updated - 2016-09-19.
[121] G. G. Kumar and V Charishma, "Design of high speed vedic multiplier using vedic mathematics techniques", *International Journal of Scientific and Research Publications*, vol. 2, no. 3, p. 1, 2012.
[122] C. S Wallace, "A suggestion for a fast multiplier", eng, *IEEE Transactions on Electronic Computers*, vol. EC-13, no. 1, pp. 14–17, 1964.
[123] A. D. Booth, "A signed binary multiplication technique", eng, *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951.
[124] G. Bewick, *Fast multiplication: Algorithms and implementation*, eng, 1994.
[125] C. S. Smith and S. Smith, "Comparison of null convention booth2 multipliers", in *CDES*, 2010.
[126] S. C. Smith, "Designing null convention combinational circuits to fully utilize gate-level pipelining for maximum throughput", in *ESA/VLSI*, 2004.
[127] L. Dadda, "Some schemes for parallel multipliers", eng, *Frequenza*, vol. 34, pp. 349–356, 1965.
[128] M. M. Kim, J. Kim, and P. Beckett, "Area performance tradeoffs in NCL multipliers using two-dimensional pipelining", in *2015 International SoC Design Conference (ISOCC)*, Nov. 2015, pp. 125–126.
[129] F. A. Parsan, J. Zhao, and S. C. Smith, "SCL design of a pipelined 8051 ALU", in *57th International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2014, pp. 885–888, ISBN: 1548-3746.
[130] Y. Yanfei, Y. Yintang, Z. Zhangming, and Z. Duan, "A high-speed asynchronous array multiplier based on multi-threshold semi-static NULL convention logic pipeline", in *9th International Conference on ASIC (ASICON)*, 2011, pp. 633–636, ISBN: 2162-7541.
[131] S. C. Smith, R. F. DeMara, J. S. Yuan, M. Hagedorn, and D. Ferguson, "NULL convention multiply and accumulate unit with conditional rounding, scaling, and saturation", *Journal of Systems Architecture*, vol. 47, no. 12,
[132] H. Zhu, Y. Zhu, C.-K. Cheng, and D. Harris, "An interconnect-centric approach to cyclic shifter design using fanout splitting and cell order optimization", eng, in *2007 Asia and South Pacific Design Automation Conference*, IEEE, 2007, pp. 616–621, ISBN: 1424406293.
[133] K. M. F. L. Kinney, *Null convention register file*, US Patent 5896541A, 1995.

[134] M. M. Kim, K. M. Fant, and P. Beckett, "Design of asynchronous risc cpu register-file write-back queue", in *2015 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, Oct. 2015, pp. 31–36.

[135] D. A. Patterson and A. Waterman, *The RISC-V reader: an open architecture atlas*. Strawberry Canyon LLC, 2017, 2017, ISBN: 9780999249109.

[136] R. Foundation. (). Risc-v specification, [Online]. Available: https://riscv.org/specifications.

[137] UC-Berkeley. (). riscv-tests, [Online]. Available: https://github.com/riscv/riscv-tests.

[138] ——, (). Spike RISC-V ISA Simulator, [Online]. Available: https://github.com/riscv/riscv-isa-sim.

[139] R. P. Weicker, "Dhrystone: A synthetic systems programming benchmark", *Communications of the ACM*, vol. 27, no. 10, pp. 1013–1030, Oct. 1984.

[140] Wiki. (2019). Radix sort, [Online]. Available: https://en.wikipedia.org/wiki/Radix_sort.

[141] ——, (). Tower of hanoi, [Online]. Available: https://en.wikipedia.org/wiki/Tower_of_Hanoi.

[142] R.-V. Cores. (). Risc-v cores, [Online]. Available: https://riscv.org/risc-v-cores/.

[143] C. Wolf. (). Picorv32 core, [Online]. Available: https://github.com/cliffordwolf/picorv32.

[144] Chips Alliance. (). Rocket Chip Generator, [Online]. Available: https://github.com/chipsalliance/rocket-chip.

[145] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The Rocket Chip Generator", EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr. 2016.

[146] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avi˘zienis, J. Wawrzynek, and K. Asanovi´c, "Chisel: Constructing hardware in a scala embedded language.", in *Design Automation Conference*, Design Automation Conference, 2012, 1216–1225.

[147] Chips Alliance. (). Chips alliance, [Online]. Available: https://chipsalliance.org/.

[148] SiFive Co., Ltd. (). Sifive, [Online]. Available: https://www.sifive.com/.

[149] UC-Berkeley. (). Chisel, [Online]. Available: https://chisel.eecs.berkeley.edu.

[150] R Sovani, "Near and sub-threshold null convention logic design for low-power digital signal processing applications", Master's thesis, 2016.

Appendix Chapter A

UNCLE Project Generation Manual

A.1 UNCLE tool introduction

The Unified NCL Environment (UNCLE) is a synthesis tool which converts a clock based behavioral Verilog file to a Null Convention Logic (NCL) Verilog gate-level net-list. UNCLE was developed by Mississippi State University in 2011. The Executable files and Manual (UNCLE.pdf) are down-loadable from their web-site. UNCLE has no usage restrictions and no limitations for any purposes. It uses general Verilog synthesis tools such as Design Compiler of Synopsys or RTL Compiler of Cadence to generate clocked Verilog gate-level net-list as an intermediate file to convert NCL base gate-level net-list. Using these synthesis tools, UNCLE generates the Verilog gate level net-list file first and converts that to the NCL net-list file. UNCLE automatically executes these synthesis steps using Python scripts.

A.2 UNCLE tool installation and setup

UNCLE tool download and setup:

Download UNCLE from the UNCLE web-site:

https://sites.google.com/site/asynctools/

Move the file where you want to install and untar the file into the directory:

```
>tar xzvf release.tgz
```

UNCLE setup:

After untar the file, you have to add uncle binary directory to your Linux Shell PATH.
In the case of TCSH shell:

```
setenv UNCLE ~/uncle
setenv IUS_HOME /opt/cadence/INCISIV102
set UNCLE_PATH = ($UNCLE/mapping/tools/bin/ $UNCLE/mapping/tools/x86_64/bin)
set path = ($UNCLE_PATH $path)
```

UNCLE tool initialization:

Once UNCLE is downloaded, execute the installation command at least once to setup and generate simulation library in your UNCLE directory (please see $UNCLE/README.txt):

```
>python install_test.py cadence
>python install_test.py synopsys
```

Once this command is successful, the tool will compile and generate a simulation library for each company's simulation tools (VCS from Synopsys, NC-Verilog from Cadence).

A.3 UNCLE Project Generation

UNCLE has many design examples inside its directory, which can be tested and simulated to test UNCLE setup and external simulation tools.

>cd $UNCLE/designs/regress

>python doregress.py up_counter cadence default.ini –syntool synopsys

This command will generate an NCL netlist of up_counter using Design Compiler of Synopsys and will simulate the net-list using NC-Verilog of Cadence.
The simulation tool will simply generate text based simulation comparison results from their example test-bench.

UNCLE Automatic Project Generation Script:
Automatic project generation script was generated using Python and the file name is batchprocess.py. This is the setup and execution sequence to use the script.
1> Move to the $UNCLE/designs directory.

>cd $UNCLE/designs

2> Copy the batchprocess.py script under $UNCLE/designs.
3> Before starting the process, generate the "files" directory under your $UNCLE/designs directory.

>mkdir $UNCLE/designs/files

4> Copy the synchronous Verilog file and NCL test-bench file to the generated $UNCLE/designs/files directory.
Test-bench files have to follow this naming convention: tb_ncl_<design name>.v
for example: tb_ncl_up_counter.v
5> Execute the script with two or three arguments. This is an example:

>python batchprocess.py up_counter clk_up_counter

The first argument is design name (basename) and the second argument is input synchronous file name (sourcename) without ".v" and the third argument is the selection of synthesis tool which is synopsys (Synopsys Design Compiler) or cadence (Cadence RTL Compiler). We can omit the last argument and the command will use Synopsys as a default synthesis tool. The script will generate NCL net-list file and copy the net-list to your $UNCLE/designs/files directory as an execution result. The result net-list file should be named "ncl_<design name>.v" for example: ncl_up_counter.v

The script automatically generates "project" directory under your $UNCLE/designs directory if you do not have that name and copy all the directories and files of $UNCLE/designs/regression and execute all the process in the generated project directory.

If you have no test-bench file, the command will generate Errors but it will still copy the NCL netlist file to your $UNCLE/designs/files directory. After that, you can generate the testbench using the new NCL netlist and can also execute the same process again with the testbench file without Errors.

This is an example command when we use the Cadence RTL Compiler synthesis tool:

>python batchprocess.py up_counter clk_up_counter cadence

The rest of the pages are "batchprocess.py" Python code:

```
#! /usr/bin/python
############################################################
# UNCLE process batch file
# Automatically generate UNCLE project and convert clocked synchronous Verilog design
# file to NCL Verilog net-list
#
# Institute: RMIT University
# Author: Matthew Myungha Kim
# Instructions:
# 1. Copy this script file under $UNCLE/designs
# 2. Before starting the process, generate the "files" folder under your
# $UNCLE/designs directory => $UNCLE/designs/files
# 3. Copy the synchronous Verilog file and NCL test-bench file to the
#   $UNCLE/designs/files directory
#         (Test-bench file must use this naming convention: tb_ncl_<design neme>.v,
#   ex) tb_ncl_up_counter.v
# 4. Execute this script with two arguments
#
# Example command: python batchprocess.py up_counter clk_up_counter
# Cadence synthesis example : python batchprocess.py up_counter clk_up_counter cadence
############################################################

import os, re, shutil, sys, subprocess
import pandas as pd

UNCLE = os.path.expandvars('$UNCLE')
DESIGN = os.getcwd()
PROJECT = os.path.join(DESIGN, 'project')

def CopySource(basename, pfilename):
        # Copy source file to the $UNCLE/designs/project/syn/rtl
        # directory from the files directory
        dest = os.path.join(PROJECT, 'syn', 'rtl', '%s.v' % basename)
        src = os.path.join(DESIGN, 'files','%s.v' % basename)
        print dest
        print src
        if os.path.exists(src):
                shutil.copyfile(src, dest)
                print "Copied the source file: %s.v from the files directory" % sourcename
        else:
                if not os.path.exists(dest):
                        print "Cannot find source and destination files"
                        sys.exit(-1) #Fail
                else:
                        print "Using the previous file in the \
                        $UNCLE/designs/project/syn/rtl directory"

        # pfile (port file) copy from files directory to the
        # $UNCLE/designs/project/map directory
        dest_pfile = os.path.join(PROJECT, 'map','%s' % pfilename)
        src_pfile = os.path.join(DESIGN, 'files','%s' % pfilename)
        if not pfilename == None:
                if os.path.exists(src_pfile):
                        shutil.copyfile(src_pfile, dest_pfile)
                        print "Copied the port file: %s from the files directory" % pfilename
                else:
                        if not os.path.exists(dest_pfile):
                                print "Cannot find source and destination port files"
                                sys.exit(-1) #Fail
                        else:
                                print "Using the previous port file in the \
                                $UNCLE/designs/project/map directory"
```

```
def CopyDest(basename):
        # Copy NCL net-list file to the $UNCLE/designs/files directory
        dest = os.path.join(DESIGN, 'files','ncl_%s.v' % basename)
        src = os.path.join(PROJECT, 'sim', 'src', 'ncl_%s' % basename, 'ncl_%s.v' % basename)
        shutil.copyfile(src, dest) #Overwrite new file
        print "Copied the result file: ncl_%s.v to the files directory" % basename

def gen_csv(basename):
        src = os.path.join(DESIGN, 'files','ncl_%s.v' % basename)
        try:
                fr=open(src, 'r')
        except:
                print('Cannot open the final netlist file for gate counting')
                sys.exit(-1)
        lines = fr.readlines()

        csvdir = os.path.join(DESIGN, 'files','csv')
        if not os.path.exists(csvdir):
                os.mkdir(csvdir)
        fw_csv=open('%s/%s.csv'% (csvdir,basename), 'w')

        pattern_g = re.compile('\s+(\w+)\s+(\W*\w+\W*\w*\W*\w*\W*\w*)\s+\(.\w+\s+\(\s+')
        for rd_one_line in lines:
                ptn_g = pattern_g.match(rd_one_line)
                if ptn_g is not None:
                        fw_csv.write('%s,%s\n'% (ptn_g.group(1),ptn_g.group(2)))
        print('Completed %s.csv file generation'% basename)

def convert(basename):
        csvdir = os.path.join(DESIGN, 'files','csv')
        df = pd.read_csv('%s/%s.csv'% (csvdir,basename), index_col=False, header=None)
        df2 = df.groupby(0).count()
        filedir = os.path.join(DESIGN, 'files')
        df2.loc[:,[1]].to_csv('%s/%s_gate_cnt.csv'% (filedir, basename), mode='a', \
        header=False)

        print'Completed gate counting'

def file_merge(basename):
        csvdir = os.path.join(DESIGN, 'files','csv')
        src = os.path.join(DESIGN, 'files','ncl_%s.v' % basename)
        dst = os.path.join(DESIGN, 'files','ncl_%s.v' % basename)
        with open(src, 'wb') as outfile:
                with open(src, 'r') as infile:
                        outfile.write(infile.read())
                with open('%s/%s.csv'% (csvdir,basename), 'r') as infile:
                        outfile.write(infile.read())
        print ('csv File Merge Completed\n')

def AddRegress(doregress, basename, sourcename, pfilename):
        prog = re.compile(r'^tests\s*=\s*(\[.+^\s*])', re.DOTALL | re.MULTILINE)
        with open(doregress, 'rt') as f:
                data = f.read()
        tests = eval(prog.search(data).group(1))
        # If the basename is already there, we're done
        if any(t[0] == basename for t in tests): #compare only the first element
                return
        ncl = 'ncl_' + basename
        if pfilename == None:
                new = [basename, [ncl, [[sourcename, ncl]], '-maxcycles 100']]
        else:
```

```
                pfilewrite = '-pfile ' + pfilename
                maxwirte = '-maxcycles 100 '+ pfilewrite
                new = [basename, [ncl, [[sourcename, ncl, pfilewrite]], maxwirte]]
        tests.append(new)
        tests = ',\n'.join('\t%r' % l for l in tests)
        tests = 'tests = [\n%s\n\t\t]' % tests
        data = prog.sub(lambda m: tests, data) #lambda = generate local function
        with open(doregress, 'wt') as f:
                f.write(data)

def SetupTestbench(basename):
        prevbasename = 'ncl_mul8x8'
        # Make a new directory under project/sim/src
        ncldest = os.path.join(PROJECT, 'sim', 'src', 'ncl_' + basename)
        nclsrc = os.path.join(PROJECT, 'sim', 'src', prevbasename)
        tbsrc = os.path.join(DESIGN, 'files')
        if not os.path.exists(ncldest):
                os.mkdir(ncldest)
        # Copy ncl test-bench file
        src = os.path.join(tbsrc, 'tb_ncl_%s.v' % basename)
        dest = os.path.join(ncldest, 'tb_ncl_%s.v' % basename)
        if os.path.exists(src):
                shutil.copyfile(src, dest)
                print "Copied the testbench file: tb_ncl_%s.v \
                from the files directory" % basename
        else:
                if not os.path.exists(dest):
                        print "Cannot find the Test-bench file"
                else:
                        print "Using the previous Test-bench file in the \
                        $UNCLE/designs/project/sim/src/ncl_%s directory" % basename
        # Copy and modify makefile - using "ncl_mul8x8 Makefile"
        dest = os.path.join(ncldest, 'Makefile')
        if not os.path.exists(dest):
                src = os.path.join(nclsrc, 'Makefile')
                # Read in source
                with open(src, 'rt') as f:
                        data = f.read()
                ncl = 'ncl_' + basename
                # Change mul8x8 to "ncl_ + basename"
                m = data.replace(prevbasename, ncl)
                # Write out
                with open(dest, 'wt') as f:
                        f.write(m)

def ModifyCdsLib(basename):
        cdslib = os.path.join(PROJECT, 'sim', 'src', 'cds.lib')
        def Read():
                prog = re.compile('DEFINE\s+(\S+)\s+(\S+)')
                with open(cdslib, 'rt') as f: # open the file as read + text mode - 'rt'
                        for line in f:
                                if line.strip() != '':
                                        name, path = prog.match(line).groups()
                                        yield name, path
        # Read in the defines
        # If we've already added basename, we're done
        defines = list(Read())
        ncl = 'ncl_' + basename
        if any(n == ncl for n, _ in defines):
                return
        # Add basename
        defines.append((ncl, os.path.join('..', 'obj', 'cadence', ncl)))
```

```
        # Now turn defines back to test
        defines = '\n'.join('DEFINE %s %s' % t for t in defines)
        # Write back out
        with open(cdslib, 'wt') as f:
                f.write(defines)

############################################################
# Process Starts Here
############################################################
# Copy "regress" to "project" if not already created
if not os.path.exists(PROJECT):
        src = os.path.join(UNCLE, 'designs', 'regress')
        shutil.copytree(src, PROJECT)
        print "Generated new project directory"
# grab the basename of the newfile we're creating
# eg. for a mul2x2.v the basename is mul2x2
args = sys.argv[1:]
basename, args = args[0], args[1:]
sourcename, args = args[0], args[1:]
# pfile for seperate ackin/ackout pins generation
pfilename = None
# Select the synthesis tool - synopsys or cadence
simtool = 'synopsys' # Default synopsys
if len(args) > 0:
        if args[0] == '-pfile':
                pfilename = args[1]
                if len(args) > 2:
                        if args[2] == 'cadence':
                                simtool = 'cadence'
                        elif args[2] == 'synopsys':
                                simtool = 'synopsys'
                        else:
                                print "Simulation Tool Argument Error - The sixth argument \
                                must be 'synopsys', 'cadence' or leave blank"
                                sys.exit(-1) # Fail
        else:
                if args[0] == 'cadence':
                        simtool = 'cadence'
                elif args[0] == 'synopsys':
                        simtool = 'synopsys'
                else:
                        print "Simulation Tool Argument Error - The forth argument \
                        must be 'synopsys', 'cadence' or leave blank - if it is not -pfile "
                        sys.exit(-1) # Fail

CopySource(sourcename, pfilename)
AddRegress(os.path.join(PROJECT, 'doregress.py'), basename, sourcename, pfilename)
SetupTestbench(basename)
ModifyCdsLib(basename)
os.chdir(PROJECT)
subprocess.call(['python', 'doregress.py', basename, 'cadence', 'default.ini', \
}'-syntool', simtool])
CopyDest(basename)
gen_csv(basename)
convert(basename)
print "All completed"
sys.exit(0) # Successful
```

Appendix Chapter B

NCL FIR Filter Design on Commercial FPGA

B.1 Introduction

Null Convention Logic (NCL) based Finite Impulse Response (FIR) Filter design is implemented on the Commercial FPGA (Altera STRATIX-V). We have generated a Band Pass filter and the filter has 100Taps to meet sharp filter specifications and real-time pipeline performance. The filter is generated by MATLAB Filter Builder and Filter Design HDL Coder tools. The tools generate Verilog or VHDL Register Transfer Level (RTL) code automatically as well as their Test-bench code. This report will present the Area and Power Analysis result and their comparison of the NCL filter design and its Synchronous count part.

B.2 FIR Filter Generation

Generation Condition:

The Filter has Band Pass specification which is 125 to 175 Hz filtering bandwidth and it has 16 KHz sampling Frequency (Figure B-1). The filter has designed for the specific audio signal processing.

Filter Specifications:

```
// Filter Specifications:
//
// Sampling Frequency   : 16 kHz
// Response             : Bandpass
// Specification        : N,Fst1,Fp1,Fp2,Fst2
// Filter Order         : 100
// First Stopband Edge  : 100 Hz
// First Passband Edge  : 125 Hz
// Second Passband Edge : 175 Hz
// Second Stopband Edge : 200 Hz
```

Figure B-1: *FIR Filter Specifications*

Figure B-2 shows us the frequency spectrum specifications for the FIR Filter. Figure B-3 has filter generation conditions. We have used Fixed Point types for Arithmetic Logic modules of the filter and the types can be efficiently implemented on the commercial FPGA or ASIC compared to the Floating Point types.

Figure B-4 explains detailed settings which are used on the MATLAB tools for the filter generation.

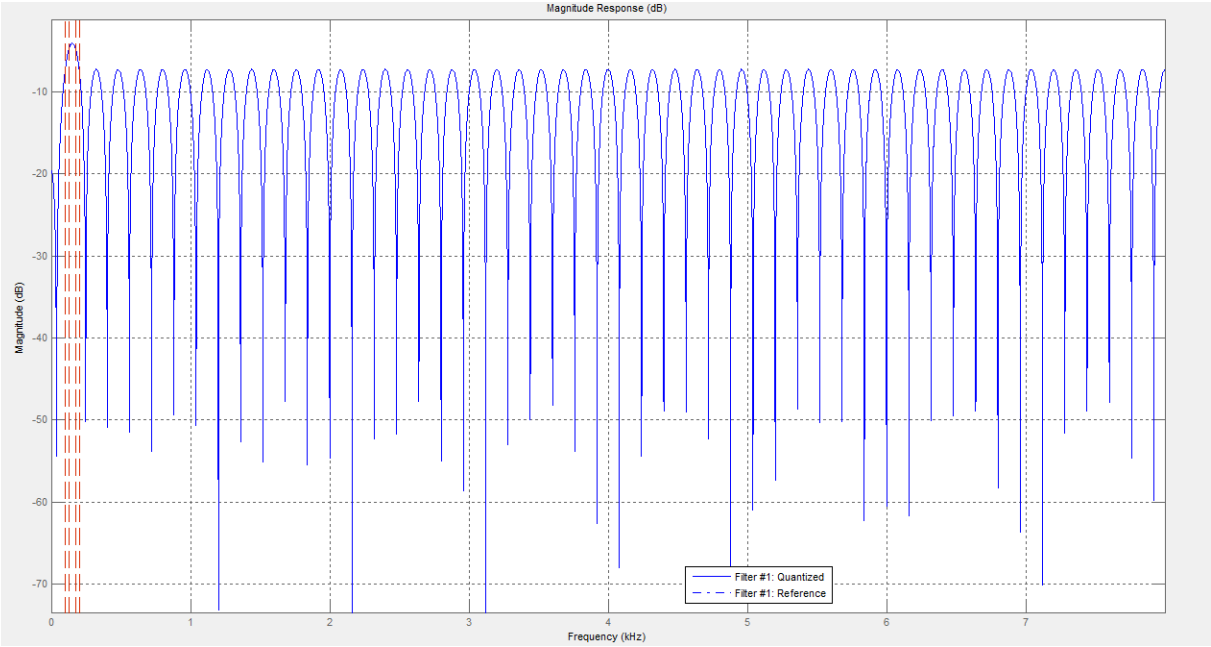

Figure B-2: *Band Pass Filter Spectrum Specification*

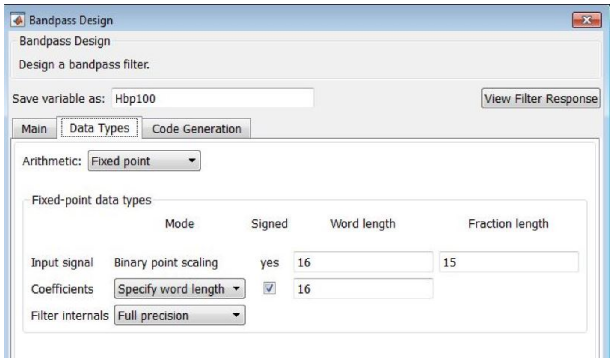

Figure B-3: *BPF Generation Condition*

```
// ---------------------------------------------------------------
// HDL Implementation    : Fully parallel
// ---------------------------------------------------------------
// Filter Settings:
//
// Discrete-Time FIR Filter (real)
// -------------------------------
// Filter Structure  : Direct-Form FIR Transposed
// Filter Length     : 101
// Stable            : Yes
// Linear Phase      : Yes (Type 1)
// Arithmetic        : fixed
// Numerator         : s16,17 -> [-2.500000e-01 2.500000e-01)
// Input             : s16,15 -> [-1 1)
// Filter Internals  : Full Precision
//   Output          : s33,32 -> [-1 1)  (auto determined)
//   States          : s33,32 -> [-1 1)  (auto determined)
//   Product         : s31,32 -> [-2.500000e-01 2.500000e-01)  (auto determined)
//   Accumulator     : s33,32 -> [-1 1)  (auto determined)
//   Round Mode      : No rounding
//   Overflow Mode   : No overflow
// ---------------------------------------------------------------
```

Figure B-4: *FIR Filter Setting*

B.3 Synchronous to NCL conversion

UNCLE:

Behavioral Verilog design of synchronous FIR Filter is converted to the NCL gate level net-list using Unified NCL Environment (UNCLE) tools. UNCLE uses general Verilog synthesis tools such as Design Compiler of Synopsys or RTL Compiler of Cadence to generate clocked Verilog gate-level net-list as an intermediate file which is converted to the final NCL base gate-level net-list.

B.4 Area Analysis Result

Area Resources Analysis and Comparison:

This paragraph explains the Area Resource Analysis of FIR Filter on the Altera Stratix-V device (Device: 5SGSMD8N1F45C2). FIR Filter 100Taps uses a number of Multipliers and Adders. When we synthesis this design using default synthesis option, the design uses 44 DSP Blocks (Table B-1) for Multipliers. The DSP blocks does not use Logic Elements (LE) because they are dedicated Hard-macro IP Blocks on the FPGA chip.

For proper Logic Resources and Power Consumption comparison between Synchronous Design and NCL Design, we have changed Altera synthesis options to use Logic Elements of their

DSP Block Balancing on the Quartus tools. When we implemented Synchronous Filter Design using DSP Blocks, the logic used 1,809 LEs with 44 DSP Blocks. Without using DSP Blocks, the design used 2,945 LEs. NCL Design which is generated by UNCLE synthesis tool used 28,577 LEs for the same Filter Design architecture (Table B-1).

The NCL design uses more than 10 times Logic Elements that is because each NCL gates are implemented using each different LEs without logic optimization on the FPGA Look Up Table (LUT). NCL logic also does not use internal Flip-Flops of Logic Elements and all the NCL logics are implemented using LUT only.

Table B-1: FIR Filter 100 Taps Fit Report

Model	Synchronous	Synchronous with DSP	NCL
Logic Utilization	2,945/262,400(1%)	1,809/262,400(1%)	28,577/262,400(11%)
Total Registers	3346	3346	0
Total Pins	52/1,064(5%)	52/1,064(5%)	101/1,064(9%)
Total DSP Blocks	0/1,9673(0%)	44/1,963(2%)	0/1,964(0%)

B.5 Simulation and Stimulus Generation

Test-Bench and Simulation:

For more accurate Power Analysis, we have used Value Change Dump (.vcd) file and the file was generated by Modelsim simulation tool. We have used same stimulus inputs for the Synchronous and the NCL designs which explains the two design styles have the same toggling requirements.

NCL Logic Transition Rate Control for Functional Simulation:

It is important generating same logic transition rate for power comparison for the two different logic architectures. We have used 50MHz clock rate for Synchronous design (Figure B-5) and we also generated same transition rate for NCL Design using 100MHz clock (Figure B-6). The 100MHz clock will generate DATA and NULL transition at the every rising edge of the clock therefore the clock generates 50MHz transition rate for NCL Design.

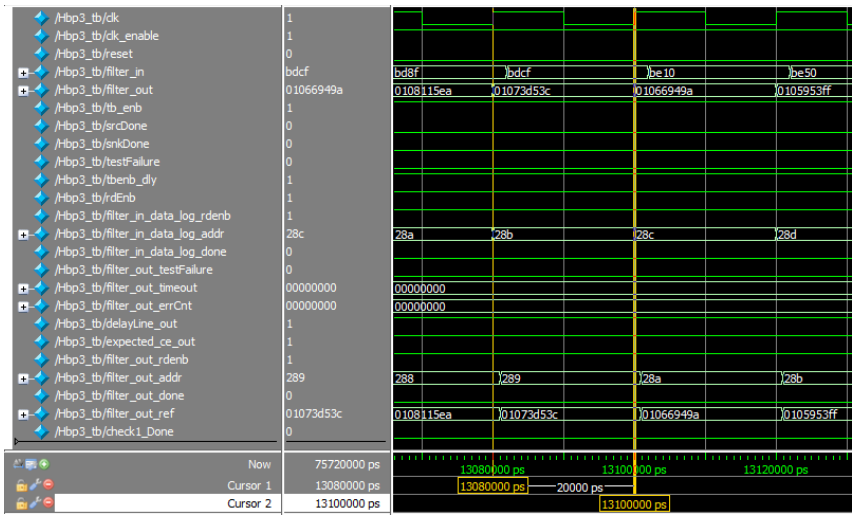

Figure B-5: *Modelsim Simulation Waveform of Synchronous Design*

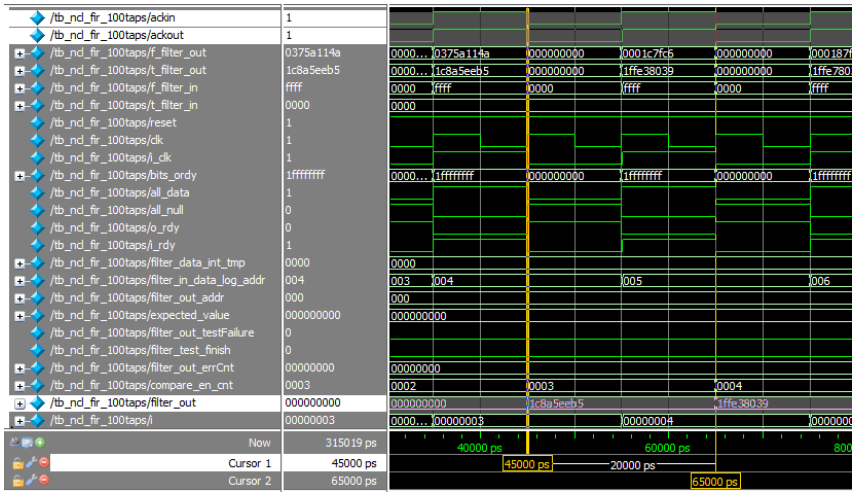

Figure B-6: *Modelsim Simulation Waveform of NCL Design*

B.6 Altera PowerPlay Power Analyser Tool

We have generated the power analysis result using Altera PowerPlay tool and the tool used .vcd files to initialize toggle rates and static probabilities during power analysis.

B.7 Power Analysis Results

Power Analysis and Comparison:

Table B-2 shows the power analysis results. The Dynamic Power of NCL Design has used around 10 times and it is similar to the logic resource usage rate. Because the two type of designs are using the same toggle rates, the dynamic power will be just multiply values of their gates counts. The two design styles has similar Static Power results.

NCL Design has no Clock Enable Block power because the logic does not use clock. NCL Design used 3 times I/O power consumption and the result was from their dual rail specifications and NULL/DATA transition style for each data input. Basically, we have to generate continuous input data for proper FIR Filter simulation. Table B-2 shows us Synchronous Design power results when we used DSP blocks for multiplier implementation. The results displays less power consumption than the design we used LEs only.

Table B-2: FIR Filter 100 Taps Fit Report

Model	Synchronous	Synchronous with DSP	NCL
Dynamic Power(mW)	112.61	86.70	1291.34
Static Power(mW)	1240.04	1239.09	1287.85
I/O Power(mW)	48.79	48.79	137.50
Total Power(mW)	1401.65	1374.58	2716.70

Appendix Chapter C

NCL Gates Verification

C.1 Introduction

This document explains Functional Simulation strategy for Null Convention Logic (NCL) Threshold Gates with NC-Verilog simulation tools of Cadence. When we design new type of gates using Schematic or Hardware Description Language (HDL) such as Verilog and VHDL, we need to make sure that the gates are functionally working properly as we intended. In this document we will show functional verification method of NCL Threshold Gates and the Gates will be compared with Reference Verilog Behavioral Gate Models. This document will present how to setup and execute proper Gates simulation especially when we use Virtuoso Composer Schematics and NC-Verilog simulation tools for Cell design.

C.2 NCL Gates

What is NCL Gates?

NCL is one type of Asynchronous digital logic design style and it is implementing Quasi-Delay Insensitive (QDI). QDI means it is almost like Delay Insensitive (DI) but has limited timing analysis. The meaning of DI is that the logic is working regardless of delay time of gates or nets. NCL gates rely on one simple timing assumption: that the feedback path for its state holding elements must be fast enough compared to the delay through the gate. To achieve these requirements, in NCL, we use input completeness function in all the combinational and sequential gates therefore most of the NCL gates, except few cases in our design, have to have input completeness functions inside their designs.

Input completeness:

Input completeness requires that the output signal of the gate may not transition to DATA until all inputs have transitioned to DATA. Similarly, the output will not transition to NULL until all inputs have reached NULL. Input Completeness is achieved by adding the value NULL to the basic logic values (TRUE and FALSE), to represent a status of “no data”. Output Data will only be valid when all input signals have transitioned from NULL to DATA. An NCL circuit consists of an interconnection of primitive modules known as M-of-N threshold gates with hysteresis. All functional blocks, including both combinational logic and storage elements, are constructed out of these same primitives.

List-up NCL Gates and Classification:

In our NCL gate design cases, we implemented 34 Threshold Gates (Table C-1). We added 7 more gates to the NCL 27 fundamental gates to implement Unified NCL Environment (UNCLE) net-list and this list was from the Simulation model of UNCLE and these 34 Gates will be basic components of the UNCLE net-list. In Table C-2 NCL Gates are classified to 6 different cases by the reset and input conditions and Table C-3 shows NCL Gates classification list for each case.

Table C-1: NCL Threshold Gates List

No	Gate Name	No	Gate Name	No	Gate Name
1	th34w32	13	th22r	25	th13
2	th54w32	14	th22s	26	th12
3	thxor0	15	th23	27	th12b
4	thxor0n	16	th54w22	28	th54w322
5	th24w2	17	th34w22	29	th14
6	th44w22	18	th44	30	th33
7	th34w2	19	th24w22	31	th33r
8	th34w3	20	th44w3	32	thand0
9	th24	21	th44w2	33	th44w322
10	th24comp	22	th33w2	34	th44w322a
11	th22	23	th34		
12	th22x8	24	th23w2		

Table C-2: Threshold Gates Classification for Simulation

Case Number	Description
Case1	2 input without reset
Case2	2 input with reset
Case3	3 input without reset
Case4	3 input with reset
Case5	4 input without reset
Case6	4 input with reset

Table C-3: Threshold Gates Classification List

Case Number	Gate Name	Case Number	Gate Name
Case 1	th22	Case 5	th34w2
	th22x8		th34w3
	th12		th24
	th12b		th24comp
Case2	th22r		th54w22
	th22s		th34w22
Case3	th23		th44
	th33w2		th24w22
	th23w2		th44w3
	th13		th44w2
	th33		th34
Case4	th33r		th54w322
Case5	th34w32		th14
	th54w32		thand0
	thxor0		th44w322
	th24w2		th44w322a
	th44w22	Case 6	thxor0n

C.3 NCL Gates Input Conditions

Monotonic conditions for each different input case:

Asynchronous digital logic, especially NCL gates are vulnerable to glitches at their inputs that can cause hazards within the internal hysteresis latch function (Figure C-1). In NCL, as long as the input transitions are monotonic, there can be no races or hazards and thus no spurious result symbols during the propagation of the monotonic wave front of correct results through the combinational expression. Maintaining monotonic behaviour and avoiding glitches mandates careful testing of the NCL circuits.

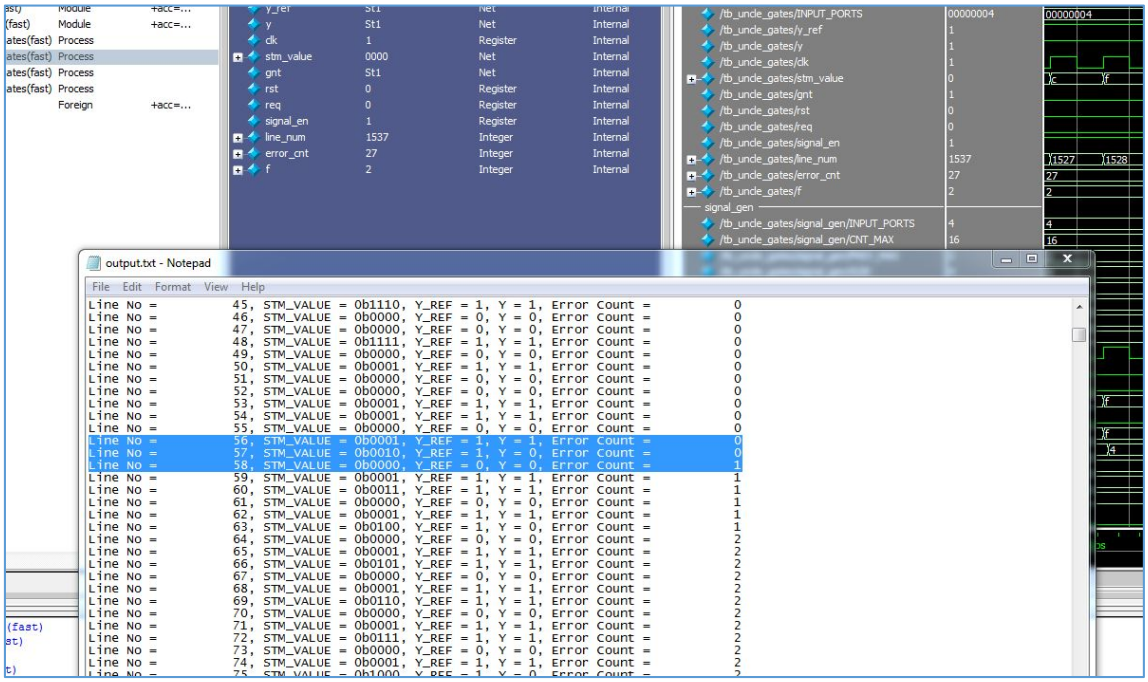

Figure C-1: NCL Gates Glitch Input and Hazard Condition on the Modelsim simulation tool

Figure C-2 and Figure C-3 show us the monotonic transition cases from NULL to DATA and DATA to NULL. Using these transition cases, we generated monotonic input stimulus files for each different cases.

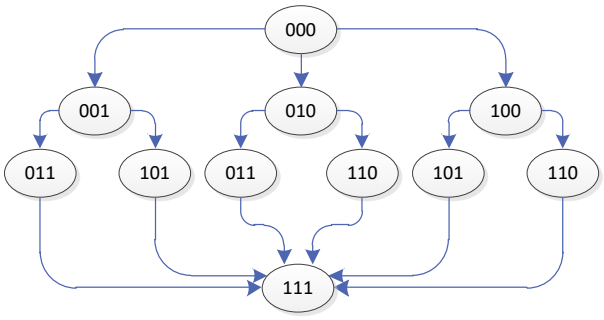

Figure C-2: Monotonic Input Value Transition, 3 input condition, 000 to 111 (NULL to DATA)

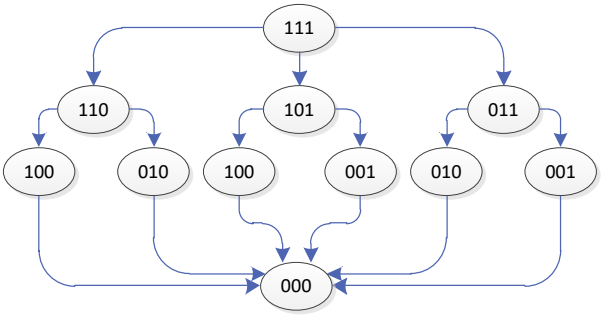

Figure C-3: Monotonic Input Value Transition, 3 input condition, 111 to 000 (DATA to NULL)

C.4 Test-bench Architecture

Test-Bench Architecture and Descriptions:

Figure C-4 shows Block Diagram of Test-bench. The test-bench has three parts; Input, Threshold Gate and Output parts. There are two types of inputs; Full Transition Stimulus Signal Generation (FTSG) and Monotonic Stimulus Signal Generation parts (MSSG). FTSG (*signal_gen.v*) generates all the possible transition input cases and it used state-machine to generate continuous input signals. MSSG (*stimulus_gen.v*) generates monotonic input signals and after calculating each monotonic cases (Figure C-2 and Figure C-3). The file read four different input stimulus files (ncl_stimul_1input.txt, ncl_stimul_2input.txt, ncl_stimul_3input.txt, ncl_stimul_4input.txt) and the files are selected by the parameter selection from the test-bench file (*tb_uncle_gates.v*) parameter setup. The text input stimulus files were manually generated after considering monotonic input conditions (Figure C-2 and Figure C-3).

Reference Threshold Gate block has Reference Threshold Gates designed by Behavioral Verilog HDL and the reference has taken from UNCLE gate simulation models. Output part executes Result Comparison and text displays including Pass/Fail decision logic. The part generates output result as a text file (output.txt, Figure C-8) automatically when the simulation has finished.

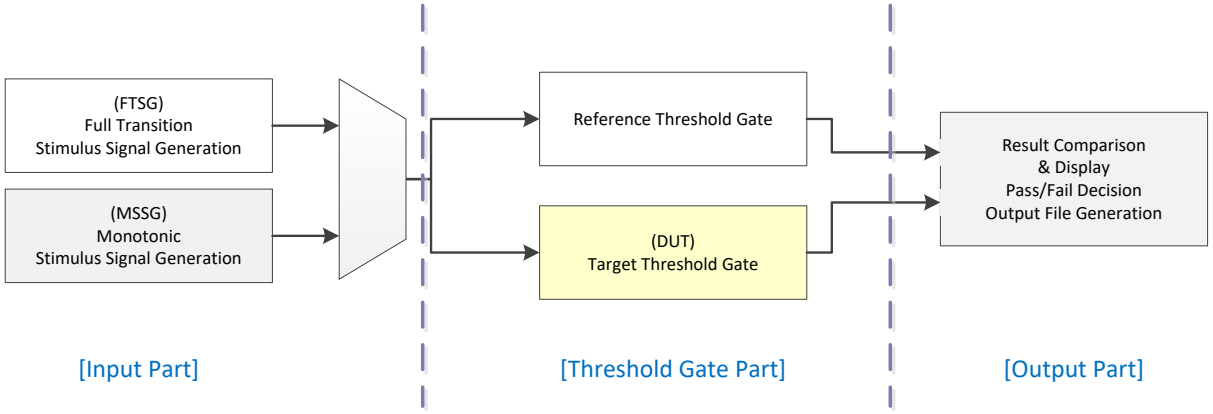

Figure C-4: Test-bench Block Diagram

Automatic Simulation:

At this stage, the test-bench has designed to select target Threshold Gates after changing test-bench file therefore we have to change 34 times to test each different gates individually. Using the automatic test script, we can test all the gates in one time.

C.5 Netlist Generation for Schematic based Threshold Gates

In the case of NCL schematic design using the Cadence Composer Schematics, we have to generate net-list file from the Virtuoso Verilog Environment for NC-Verilog Integration tool for Functional Simulation. The tool automatically generates NC-Verilog execution scripts files and Test-bench files for the Functional simulation and generates net-list using Functional View file of each gates. Unfortunately our Circuits Multi-Project (CMP) 28nm PDK library does not have Functional View of the *pfet* and *nfet* Transistor models therefore automatic test on the Integration tool will not be successful. In our case, we have to copy the net-list file to our own NC-Verilog test directory and have to execute the simulation separately.

From the NC-Verilog Integration window, execute Initialize Design, Generate Net-list in order. The tool will generate net-list file under the *th22_run1/ihnl/cds1* directory in this test case. Figure C-6 shows Schematic design of TH22 gate on the Cadence *Composer Schematics* tool.

In our case, we copied the net-list file to our NC-Verilog test directory and changed file name to *th22.v*. Figure C-5 shows detailed net-list of *th22.v* and it is a simply textual version of the schematic.

```
// Library - static, Cell - th22, View - schematic
// LAST TIME SAVED: May 23 15:00:43 2014
// NETLIST TIME: May 23 15:01:15 2014
`timescale 1ns / 1ns

module th22 ( y, a, b );

output  y;

input  a, b;


specify
    specparam CDS_LIBNAME  = "static";
    specparam CDS_CELLNAME = "th22";
    specparam CDS_VIEWNAME = "schematic";
endspecify

pfet_b  P4 ( .b(cds_globals.vdd_), .g(a), .s(cds_globals.vdd_),
     .d(net24));
pfet_b  P3 ( .b(cds_globals.vdd_), .g(y), .s(net24), .d(net15));
pfet_b  P2 ( .b(cds_globals.vdd_), .g(b), .s(cds_globals.vdd_),
     .d(net24));
pfet_b  P1 ( .b(cds_globals.vdd_), .g(b), .s(net35), .d(net15));
pfet_b  P0 ( .b(cds_globals.vdd_), .g(a), .s(cds_globals.vdd_),
     .d(net35));
nfet_b  N4 ( .d(net22), .g(b), .s(cds_globals.gnd_),
     .b(cds_globals.gnd_));
nfet_b  N3 ( .d(net22), .g(a), .s(cds_globals.gnd_),
     .b(cds_globals.gnd_));
nfet_b  N2 ( .d(net15), .g(y), .s(net22), .b(cds_globals.gnd_));
nfet_b  N1 ( .d(net34), .g(a), .s(cds_globals.gnd_),
     .b(cds_globals.gnd_));
nfet_b  N0 ( .d(net15), .g(b), .s(net34), .b(cds_globals.gnd_));
inv I8 ( y, net15);

endmodule
```

Figure C-5: TH22 netlist file

Verilog Switch-Level Modelling:

In Verilog-HDL, transistors are functionally expressed as a *Switch* which is named *Switch-Level Modelling*. For this functional simulation of the transistor based Schematic design, we have to use Verilog Switch-Level model. It is totally functional model.

We added three model files. The *cds_globals.v* file has VDD and GND sources of Gates and the *nfet_b.v* and *pfet_b.v* files have Verilog-HDL Switch-Level Models. We have used *tranif0* for pmos FET and *tranif1* for nmos FET.

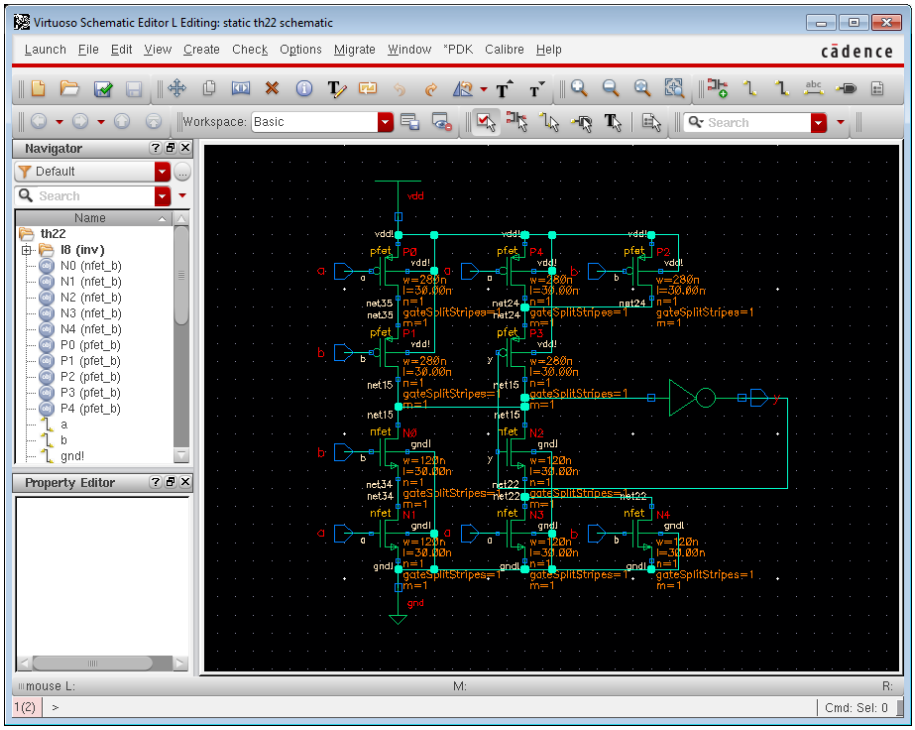

Figure C-6: TH22 Gate Schematic

C.6 NC-Verilog Simulation and Result Analysis

Simulation Results and Analysis:

In this document, we are testing TH22 gate as an example. Change the parameters on the *tb_uncle_gates.v* properly for your target gate. In this case, we changed INPUT_PORT to 2 and RESET_PORT to 0. And then, Set-up the tool environment scripts and execute NC-Verilog script file (run.sh).

$./run.sh

The run.sh script executes this command:

ncverilog +nc64bit +gui -access rwc -f netlist_files.f

netlist_files.f has the Verilog file list for simulation.

This script will automatically execute NC-Verilog and will open Simvision windows.

Select "tb_uncle_gates" on the Design Browser and select all signals what you want to see on the Waveform window (Figure C-7). Click on right mouse button and select "Set to Target" ⇒ "Waveform Window". It will open Waveform window. Go to Console – Simvision and type run 100us.

nc_sim>run 100us

The NC-Verilog simulation will be executed and the output.txt file will be generated on your project directory. Waveform window will show you all the waveforms. The test result will be displayed on your Simvision console window (Pass/Fail).

The output.txt file will show you the input conditions and output of comparison result between reference Gate and Schematic design net-list.

Figure C-7 and Figure C-8 show the test result.

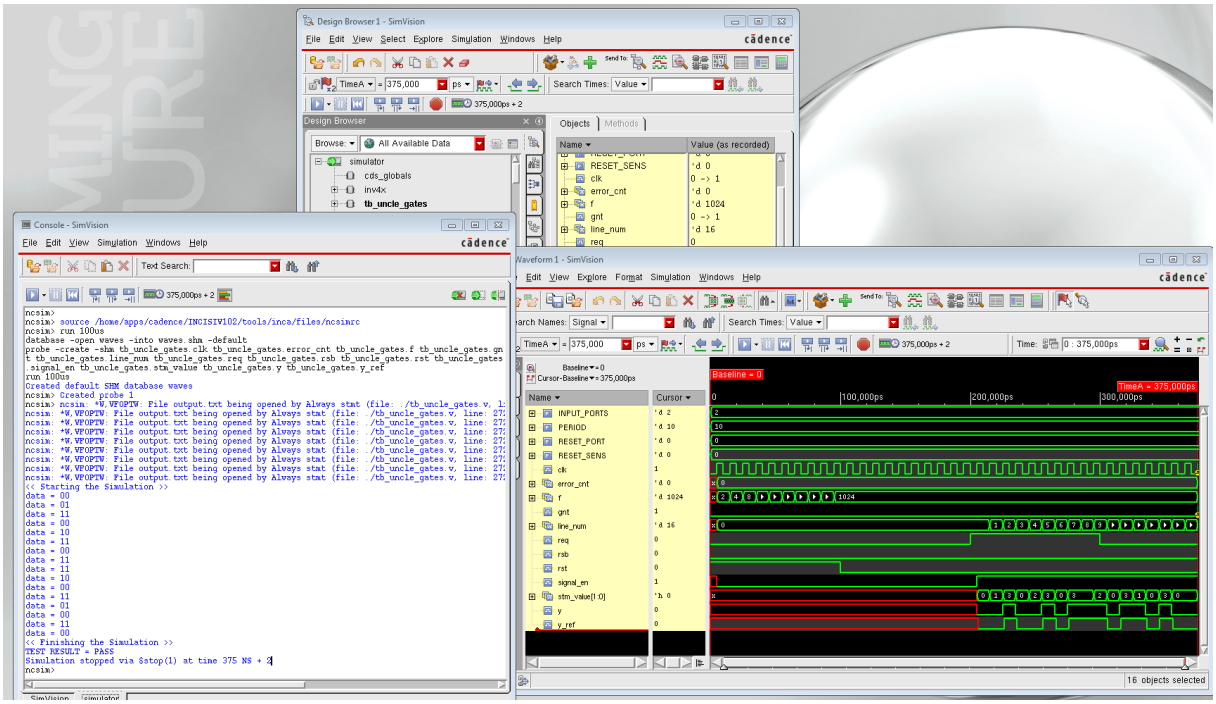

Figure C-7: NC-Verilog Execution

output.txt

```
Line No =          0, RSB = 0, STM_VALUE = 0b00, Y_REF = 0, Y = 0, Error Count =          0
Line No =          1, RSB = 0, STM_VALUE = 0b01, Y_REF = 0, Y = 0, Error Count =          0
Line No =          2, RSB = 0, STM_VALUE = 0b11, Y_REF = 1, Y = 1, Error Count =          0
Line No =          3, RSB = 0, STM_VALUE = 0b00, Y_REF = 0, Y = 0, Error Count =          0
Line No =          4, RSB = 0, STM_VALUE = 0b10, Y_REF = 0, Y = 0, Error Count =          0
Line No =          5, RSB = 0, STM_VALUE = 0b11, Y_REF = 1, Y = 1, Error Count =          0
Line No =          6, RSB = 0, STM_VALUE = 0b00, Y_REF = 0, Y = 0, Error Count =          0
Line No =          7, RSB = 0, STM_VALUE = 0b11, Y_REF = 1, Y = 1, Error Count =          0
Line No =          8, RSB = 0, STM_VALUE = 0b11, Y_REF = 1, Y = 1, Error Count =          0
Line No =          9, RSB = 0, STM_VALUE = 0b10, Y_REF = 1, Y = 1, Error Count =          0
Line No =         10, RSB = 0, STM_VALUE = 0b00, Y_REF = 0, Y = 0, Error Count =          0
Line No =         11, RSB = 0, STM_VALUE = 0b11, Y_REF = 1, Y = 1, Error Count =          0
Line No =         12, RSB = 0, STM_VALUE = 0b01, Y_REF = 1, Y = 1, Error Count =          0
Line No =         13, RSB = 0, STM_VALUE = 0b00, Y_REF = 0, Y = 0, Error Count =          0
Line No =         14, RSB = 0, STM_VALUE = 0b11, Y_REF = 1, Y = 1, Error Count =          0
Line No =         15, RSB = 0, STM_VALUE = 0b00, Y_REF = 0, Y = 0, Error Count =          0
Line No =         16, RSB = 0, STM_VALUE = 0b00, Y_REF = 0, Y = 0, Error Count =          0
```

Figure C-8: output.txt File of TH22 Gate Test

The simulation models and test-bench files for this Appendix-C are shared on the author's GitHub public repository:
`https://github.com/MatthewMyunghaKim/NCL_Gates_Functional_Simulation`